Department of Computer Science
University of Fribourg (Switzerland)

Defence Mechanisms against Vulnerabilities in Network Protocols and Risk Assessment of Data Packets

Doctoral Thesis

Submitted to the Faculty of Science of the University of Fribourg to obtain the degree of Doctor Scientiarum Informaticarum

by

InSeon Yoo

Thesis No.1520
University of Fribourg
2006

UNIVERSITY OF FRIBOURG

FACULTY OF SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

Doctor Scientiarum Informaticarum

Defence Mechanisms against Vulnerabilities in Network Protocols and Risk Assessment of Data Packets

by InSeon Yoo

Accepted by the Faculty of Science of the University of Fribourg (Switzerland) as recommend by the following professors:

- **Prof. Béat Hirsbrunner** (President of the examination committee)

  Department of Computer Science, University of Fribourg, Switzerland

- **Prof. Jan HP Eloff** (external examiner)

  Department of Computer Science, University of Pretoria, South Africa

- **Prof. Stephanie Teufel** (internal examiner)

  iimt (international institute of management in telecommunications), University of Fribourg, Switzerland

- **Prof. Ulrich Ultes-Nitsche** (thesis supervisor)

  Department of Computer Science, University of Fribourg, Switzerland

Fribourg, 26 May 2006

Thesis Supervisor: Prof. Ulrich Ultes-Nitsche Faculty Dean: Prof. Marco Celio

UNIVERSITY OF FRIBOURG

ABSTRACT

FACULTY OF SCIENCES
DEPARTMENT OF COMPUTER SCIENCE

Doctor Scientiarum Informaticarum

by InSeon Yoo

Currently, the Internet is not a privilege for specific groups any more. The wide coverage of the Internet brought lots of risks as well as convenience. Network security problems are the familiar loss of confidentiality, integrity and availability. One of the solution technologies is a defence system to protect network-connected resources, which is called a firewall. The more serious network threats are, the more important the role of the firewall is. As high technologies have developed, the ways of attacks are getting even more serious. To prevent new attacks, new defence mechanisms also need to be developed.

This thesis looks for defence mechanisms against network attacks, which use vulnerabilities in network protocols, and risk assessment of data packets, then apply them to network security systems, in particular, to a firewall system, which is called Janus. The start of this research is based on the MPhil thesis, "An Intelligent Firewall Architecture Model to Detect Internet-Scale Virus Attacks". With the concept of an intelligent firewall in mind, research on potential technologies applicable to Janus has been conducted. The goal of this research is to extend the abilities of packet-filtering firewalls aiming to reduce possible problems and attacks by improving firewall technologies. Janus is an adaptive firewall architecture based on a packet-filtering firewall, which can deal with protocol anomaly detection and verification, and email classification. Furthermore, Janus can conduct virus detection in attached files without the use of virus signatures. The non-signature-based virus detection approach currently is capable of detecting 84% of the virus-infected files in the sample set, which includes polymorphic and encrypted viruses. At the moment, the false positive rate is 30%. The combination of the classical virus detection technique for known viruses and this SOM-based technique for unknown viruses can help systems even more secure. The contribution of this work is to propose various approaches for building defence mechanisms, and developing the adaptive firewall model further based on the proposed defence technologies.

UNIVERSITÄT FREIBURG (CH)

ZUSAMMENFASSUNG

MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
DEPARTEMENT FÜR INFORMATIK

Doctor scientiarum informaticarum (Dr. sc. inf.)

vorgelegt von InSeon Yoo

Heutzutage ist das Internet kein Privileg mehr für eine bestimmte Personengruppe. Die weite Verbreitung des Internets brachte viele Risiken wie auch Annehmlichkeiten. Probleme der Netzwerksicherheit sind der bekannte Verlust von Vertraulichket, Inegrität und Verfügbarkeit. Eine Lösungstechnologie ist ein Verteidigungssystem, das mit dem Netzwerk verbundene Resourcen schützt, die Firewall. Je grösser Netzwerkbedrohungen werden, umso wichtiger wird die Rolle der Firwall. Durch die weiterentwickelte Technologie werden die Angriffe immer bedrohlicher. Um vor neuen Angriffen zu schützen müssen, neue Verteidigungsmechanismen entwickelt werden.

Die vorliegende Doktorarbeit untersucht Verteidigungsmechanismen gegen Netzwerkangriffe, die Schwachstellen in Netzwerkprotokollen ausnutzen, und betrachtet Risikobewertung von Datenpaketen, um sie dann auf Netzwerksicherheitssysteme anzuwenden, im besonderen auf ein Firewallsystem, das Janus genannt wird. Der Anfang dieser Forschung basiert auf der MPhil Arbeit "An Intelligent Firewall Architecture Model to Detect Internet-Scale Virus Attacks". Mit dem Konzept einer intelligenten Firewall im Auge wurde an möglichen Technologien geforscht, die auf Janus anwendbar sind. Ziel dieser Forschung ist es, die Fähigkeiten einer paketfilternden Firewall auszubauen, um mögliche Probleme und Angriffe durch verbesserte Firewalltechnologien zu reduzieren. Janus ist eine adaptive Firewalltechnologie, die auf einer paketfilternden Firewall basiert und die mit Protokollanomalieerkennung, Verifikation und E-Mail-Klassifikation ausgestattet ist. Weiterhin kann Janus Virenerkennung in Dateien, die einer E-Mail angehängt sind, durchführen, ohne dabei Virensignaturen zu verwenden. Der nichtsignaturbasierte Virenerkennungsansatz ist derzeit in der Lage, 84% von vireninfizierten Dateien in einer Stichprobe zu erkennen, die auch polymorphe und verschlüsselte Viren enthält. Die *False Positives* Rate liegt im Moment bei 30%. Die Kombination klassischer Virenerkennungsverfahren mit dieser SOM-basierten Technik für unbekannte Viren kann dazu beitragen, Systeme noch sicherer zu machen. Der Beitrag dieser Doktorarbeit ist es, verschiedene Verteidigungsmechanismen anzubieten und mit diesen das adaptive Firewallmodell weiterzuentwickeln.

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

IDS	Intrusion Detection Systems
OBDD	Ordered Binary Decision Diagram
SOM	Self-Organizing Map
DNS	Domain Name Service
RFC	Request for Comments
BCc	Blind Carbon copy
RPC	Remote Procedure Call
IP	Internet protocol
ECN	Explicit Congestion Notification
DF	Don't Fragment
MTU	Maximum Transmission Unit
MF	More Fragments
TTL	Time to live
ICMP	Internet Control Messaging Protocol
UDP	User Datagram Protocol
TCP	Transmission control protocol
URG	Urgent
ACK	Acknowledgement
RST	Reset
FIN	Final
MSS	Maximum Segment Size
WS	Window Scale
SACK	Selective Acknowledgement
TS	TimeStamps
MSL	Maximum Segment Lifetime
SDL	Specification and Description Language
ITU	International Telecommunication Union
CEFSM	Communicating Extended Finite State Machine
FSM	Finite State Machine
EFSM	Extended Finite-State Machine
ISNs	Initial TCP Sequence Numbers
UBEs	Unsolicited Bulk Emails, spams

M	a group of malicious packets
E	specific evidence about malicious packets
SMTP	Simple Mail Transfer Protocol
CC	Carbon Copy
H	the header field of a mail packet
Fr	the sender field of a mail packet
To	the recipient field of a mail packet
EF	the attachment field of a mail packet
BDDs	Binary decision diagrams
INF	If-then-else Normal Form
NewEXE	portable executable format, NE, PE, LE, LX
DOS	Denial of Service
Umatrix	unified distance matrix
Virus mask	a particular pattern which signals the presence of a virus in SOM projection
DS	DOS stub
PE	NewEXE header
PR	program code
VS	virus code
DiffServ	Differentiated Services

Acknowledgements

Most of all, I shall give all glory, honor and thank to my heavenly Father, Jesus Christ, my Lord and Saviour with my faith and obedience. I have never been alone and by the grace of Jesus Christ, I am what I am. I want Him to reap His full inheritance from my life. This work could be a start for it.

By the grace of God I am what I am,
and His grace toward me did not prove vain;
but I labored even more than all of them,
yet not I, but the grace of God with me.
- 1 Corithians 15:10.

Then there are a few people I would like to thank, since I do not often get the chance. First, my parents - they made me as I am, I just hope to be worthy of their efforts. Next my supervisor, Prof. Dr. Ulrich Ultes-Nitsche, who spared no effort to ensure that I have everything I needed and then special thanks to April Isadora Lloyd who encourage me to have confidence in my English. Finally, Lucette & Danilo Hasler, Sr. Lorena and all the others who gave their time, love and energy.

He knows the way that I take;
when he has tested me, I will come forth as gold.
- Job 23:10.

InSeon Yoo
Fribourg in Switzerland
November 2005

To my friend, Jesus Christ and my parents. . .

Chapter 1

Introduction

'I wish it need not have happened in my time!', said Frodo.
'So do I', said Gandalf, 'and so do all who live to see such times.
But that is not for them to decide.
All we have to decide is what to do with the time is given us.'
- The Lord of The Rings, J.R.R. Tolkien

Currently, there are many different types of network security systems e.g., intrusion detection systems (IDS), anti-virus systems, firewalls, routers, proxies and so on. Often these techniques are employed together. Since any organization connected to the Internet deployed already some kind of routers, and most routers have at least simple packet filtering capabilities, routers are often used in addition or instead of more complex firewall products. However, routers and many other simple packet filters lack good packet handling ability for defence against network attacks which use vulnerabilities of network protocols and mechanisms. Therefore, various defence mechanisms need to be held in depth, and risk assessment of data packets must be established in the defence mechanisms.

1.1 Defence Mechanism and Network Protocols

There are many other components in the TCP/IP protocols for managing communications and providing higher-level services. Most of them were developed in the days when the network had only trusted hosts, and security was not a concern [Anderson(2001b)]. The fundamental problem of the lower-level TCP/IP protocols is that there is no real authenticity or confidentiality protection in most mechanisms. Furthermore, there are not only security problems — e.g., TCP sequence number prediction, abuse of the routing mechanisms and protocols, and so forth — in the TCP/IP protocol suite (details can be found in [Bellovin(1989)]), but also implementation problems — e.g., no initial

slow start, inconsistent retransmission, extra additive constant in congestion avoidance, excessively short keepalive connection timeout, insufficient interval between keepalives, window probe deadlock, stretch ACK violation, retransmission sends multiple packets, failure to send FIN notification promptly, failure to send a RST after half duplex close, failure to RST on close with data pending, and so on — in the TCP protocol (known implementation problems in detail are reported in RFC 2525 [V. Paxson and Volz(1999)]). Therefore, defence mechanisms must overcome these problems altogether, which can be abused by attackers.

1.2 Likelihood of Attacks

The mathematical concept of probability is inadequate to express our mental confidence or diffidence in making such inferences, and that the mathematical quantity which appears to be appropriate for measuring our order of preference among different possible populations does not in fact obey the laws of probability. To distinguish it from probability, R. A. Fisher has used the term "likelihood" to designate this quantity [Fisher(1925)].

In this thesis, the likelihood of attacks is defined as our confidence to have certain occurrence of specific attacks and attackers' preference to use certain patterns of attacks. We look at the likelihood that a particular threat [1] using a specific attack, will exploit a particular vulnerability of a system that results in an undesirable consequence, which is called risk [2]. The likelihood is not a probability, but an estimate of the threat potential of each adversary [3] group to cause an undesired event at information resources. The relative threat potential is based on characteristics of the adversary group and the information resources. The vulnerabilities in network protocols still exist as long as the structure of network and network protocols are not changed completely. As technologies are getting better, the attack skills are also getting smarter in every way. Hence, the likelihood of attacks using vulnerabilities in network protocols and mechanisms are still high although new technologies to detect or security systems' abilities are developed significantly. Therefore, we need intelligent defence mechanisms.

[1] Def. from [NSFISSI(1997)]. Any indication, circumstance, or event that can cause the loss of, damage to, or the denial of an asset.

[2] Def. from [NSFISSI(1997)]. (Likelihood of attack) * (Consequence) * (1 - System Effectiveness) = RISK.
The likelihood of a successful attack is the probability that an adversary would succeed in carrying out an attack.

[3] Def. from [NSFISSI(1997)]. Any entity that conducts, or has the capability and intention to conduct, activities detrimental to interests or assets.

1.3 Risk Assessment of Data Packets

In this thesis, the risk assessment of data packets is defined as that it is about understanding likely threats to network systems and the process of determining whether proposed or existing defence mechanisms are adequate to protect information resources from the threats. The threats to network systems are considered either using vulnerabilities in network protocols and mechanisms or using common network mechanisms with malicious code in data packets. The former can be active attacks including typical denial service attacks, – e.g., ping attacks, SYN flood attacks, land attacks and tear drop attacks – IP spoofing, spams (including phishing [Wikipedia(2005)]), and denial of service Internet worms. On the other hand, the latter can be passive attacks using social engineering methods including viruses and Internet worms.

Various existing security systems have specific defence mechanisms against specific attacks. In this thesis, firewalls are the main concern for risk assessment of data packets. Since firewalls are not smart enough to protect information resources from the abovementioned threats, this thesis proposes defence mechanisms to improve firewall technologies. Among passive attacks, Internet worms are focused on the propagation feature using email, thus the detection of viruses, Internet worms and spams is considered in an email defence mechanism. With these threat models, defence mechanisms are developed and applied to an adaptive packet-filter firewall, Janus.

1.4 Exclusions

This thesis is not concerned with flaws in particular implementations of the protocols, such as those used by Internet worms [Spafford(1988), Seeley(1988), Eichin and Rochlis(1988)]. Rather, generic problems of the protocols themselves are discussed, and various requirements for network defence systems are addressed. Careful implementation techniques can alleviate or prevent some of these problems. Generic Internet protocols will be discussed. Neither is this project concerned with physical eavesdropping, nor with altered or injected messages. This thesis discusses such problems only as far as they are facilitated or possible because of protocol problems. For the most part, there is no discussion here of vendor-specific protocols. This thesis does discuss some problems with Berkeley's protocols, since these have become de facto standards for many vendors, and not just for UNIX systems.

1.5 Reader's Guide

The rest of the PhD thesis is organised as follows. Chapter 2 contains motivation of this project, research aims and its objectives. It also describes the requirements to assess

data packets in firewalls and the proposed firewall architecture. In addition to these, the current state of virus detection is mentioned.

Chapter 3 briefly addresses analysis of vulnerabilities in network protocols and mechanisms. There are protocol anomaly-based attacks and widespread malicious code, especially through email mechanism.

Chapter 4 looks at protocol anomaly detection and verification. It also describes the threat models of protocol anomaly-based attacks and summarizes specific requirements of network protocol anomaly detection. Then a TCP runtime verification model is proposed taking into account current problems of TCP state transitions and generation of steps in the TCP verification model is presented. Furthermore, SDL modelling is presented for prototyping an application and defensive measures against protocol anomaly-based attacks.

Chapter 5 introduces email classification for risk assessment. It includes email viruses and worms, which spread via emails. A Naive Bayesian classifier and a reduced ordered binary decision diagram (OBDD) representation are used for email classification.

Chapter 6 presents virus visualization and recognition using Self-Organizing Maps (SOMs). It briefly discusses SOM algorithms and processes, and describes the application of SOM to detect virus patterns, which is discussed in Chapter 7, where application cases such as the Janus firewall system and Janus VirusDetector are introduced. The applications' implementation is outlined as well as experiments conducted as part of the research method and their results. Then these results with regard to the research's objectives are discussed and evaluated.

Finally, Chapter 8 draws conclusions with a summary of the major shortcomings of the Janus project and a brief discussion of future research.

Chapter 2

Motivations

If you don't defend your rights,
you lose them by attrition.
- Lawrence Ferlinghetti

2.1 Research Objective

This research aims to develop cost-effective defence mechanisms and apply those mechanisms to a packet-filter firewall in order to improve the firewall ability to defend against network attacks and to manage risk assessment of data packets efficiently. As a result, this research can also improve firewall technologies. Another motivation is to reduce risk through preventing some malicious packets e.g., email viruses from entering the secure network. Therefore, this project also seeks to answer the questions considering risk assessment of data packets: "how to detect worms/viruses, which are replicated via emails, at the level of a firewall without cooperation with an anti-virus server?", "how to detect email viruses without knowing their signatures?", "how to determine the probability whether the mail is abnormal?", and "how to detect virus patterns in virus-infected files?". Theoretically, it is impossible to generate a program, which can solve the general virus detection problem. This theorem has been proved in different ways in the literature. Since the general virus detection problem is not solvable, we have to reduce the problem.

To achieve the research objectives, specific requirements of network protocol usage and malicious code propagation via email has been identified, run-time verification model, email classificaiton model and virus recognition have been designed, and a prototype of a packet-filter firewall has been implemented as a proof of concept.

2.2 Increasingly Serious Attacks

Internet viruses include file viruses, file worms, and network worms. These Internet viruses are being spread via systems' security holes, emails, messengers, etc. A virus is a piece of code that adds itself to other programs and cannot run independently. As Microsoft Windows became popular, Windows viruses and Windows-application-derived viruses using Visual Basic for Applications (VBA) spread widely. A common way of Windows virus dissemination is through emails. In addition, a worm is a program that can run by itself and propagate a fully working version of it to other machines. A network worm is a worm, which copies itself to another system by using common network facilities, and causes execution of the copy on that system. A recent serious attack was Code Red. The Code Red worm is a malicious self-propagating code [CERT(2002a)] that spreads surreptitiously through a hole in certain Microsoft software. The Code Red, which left computers open to hijacking, has caused a lot of traffic being sent, clogging the bandwidth on the Internet. An infected system will show an increased processor and network load. The worm could easily permit hackers to take control of hundreds of thousands of infected machines.

The biggest impact of these worms is that their propagation creates a DOS attack in many parts of the Internet, because of the huge amount of traffic generated. DOS attacks can interrupt services by flooding networks or systems with unwanted traffic. A service will be denied, because the network or system is overwhelmed. Distributed systems based on the client/server model have become increasingly popular. Therefore Distributed Denial of Service (DDOS) attacks are also getting escalated. In an DDOS, an attacker controls a number of handlers. A handler is a compromised host with a special program running on it. Each handler is capable of controlling multiple agents. An agent is a compromised host, which is responsible for generating a stream of packets that is directed towards the intended victim.

2.3 Classification and Recognition

Classification and recognition are considered in this thesis. We can give the following situation: we may be given a set of observations with the aim of establishing the existence of classes or clusters in the data. Or we may know for certain that there are so many classes, and the aim is to establish a rule whereby we can classify a new observation into one of the existing classes.

The task of classification could cover any context in which some decision or forecast is made on the basis of currently available information, and a classification procedure is then some formal method for repeatedly making such judgments in new situations.

If we create some classifier, the classifier should be considered of accuracy, speed, comprehensibility and time to learn [D. Michie and (Eds)(1994)].

1. Accuracy. There is the reliability of the rule, usually represented by the proportion of correct classifications, although it may be that some errors are more serious than others, and it may be important to control the error rate for some key class.

2. Speed. The speed of the classifier is a major issue in real critical circumstances. 90% accurate may be preferred over one that is 95% accurate if it is 100 times faster.

3. Comprehensibility. If it is an administrator that must apply the classification procedure, the procedure must be easily understood else mistakes will be made in applying the rule. It is important also that the administrators believe the system.

4. Time to Learn. Especially in a rapidly changing environment, it may be necessary to learn a classification rule quickly, or make adjustments to an existing rule in real time.

The problem concerns the construction of a procedure that will be applied to a continuing sequence of cases, in which each new case must be assigned to one of a set of predefined classes on the basis of observed attributes or features.

The other is Pattern Recognition. There are many kinds of patterns; visual patterns, temporal patterns, logical patterns. Using a broad enough interpretation, we can find pattern recognition in every intelligent activity. No single theory of pattern recognition can possibly cope with such a broad range of problems. There are several models, statistical pattern recognition, syntactic or structural pattern recognition, knowledge-based pattern recognition and so on. These pattern recognitions could be viewed as a classification.

However, both of them are defined and used like this in this thesis: classification is based on finding proper information and establishing links between data, on the other hand, recognition is based on making a decision about the information after classifying data.

2.4 Faces vs. Packets

What is the feature of packets carrying virus codes? Compared with anti-virus systems, firewalls only deal with packets. In order to detect virus packets, we need to know the characteristics of virus packets. Assessment of packets has quite similar aspects like that with faces. Faces have images or specific characteristics like distance from nose to chin to identify a person. As Figure 2.1 shows, we can see several faces, nobody

FIGURE 2.1: Face models, all different faces, but still have common features

is the same, but everybody has common features, such as hair, eyes, one nose, one mouth and ears. Some pictures are collected from the FBI fugitive database. In terms of likelihood of attacks, the FBI keeps fugitive information, since they can use this information for several purposes to detect criminals. For example, they can sketch the figure of a criminal according to this information. So, what kind of information is useful to them? For example, size of eyes, distance between two eyes, colour of hair, eigenvector of faces, and so on. Likewise, we need a function helping us to estimate, whether or not packets are malicious and can cause undesired events. In addition, to reduce the undesired events, assessment of packets is necessary.

2.5 Current Network Security Systems' Problems

DOS attacks are easy to perpetrate and almost impossible to defend against even whilst firewalls and Intrusion Detection Systems (IDSs) are installed. Even if the Intrusion Detection Systems generate alerts, log packets, send emails, and call pagers, the attacker could still get in, and by the time somebody could respond the damage would be done. A malicious attacker could spoof attacks from many sources and effectively deny everybody access to the server. This is considered to be an unacceptable risk. A firewall would be of no help either. The web server has to remain available to the public and the vulnerability is in the web server software such as IIS. A firewall has no way of determining if a request being sent to a web server is benign or malicious. While the firewall could stop traffic to ports that do not need to be publicly accessible, it is useless in this situation. Increasingly complex security scenarios and incorrect configurations contribute to a firewall's inability to provide gateway security. A firewall is also only able to deal with traffic that passes through the firewall, with all internal traffic completely unchecked.

When a virus associated with DOS spreads through the Internet, virus scanning proxy

servers and IDSs or Firewalls must cooperate to prevent these attacks. Although anti-virus servers and software have served users well for a lengthy period, today's fast-paced technology means that viruses travel much faster than signature updates can keep up with. This kind of software often relies on databases containing these virus signatures, which catch and define viruses. It is therefore essential to ensure that the database of signatures is as up-to-date as possible. This implies that a mechanism guarantees that the latest signatures are updated, as and when new viruses are detected. Therefore a process of automatic updating of signatures, a built-in feature found in most anti-virus products, as well performing the necessary upgrade maintenance are critical actions. These are steps that cannot be left up to human processing, but must be automated to be kept up-to-date at all times. This type of solution has become vital due to the different ways that a virus can enter an organisation. This means that protection is needed at each of the levels, stopping a virus where it enters rather than having to clean up after it has spread.

2.6 Requirements To Assess Data Packets In Firewalls

2.6.1 Data Packet Detection

As social engineering attacks and security-vulnerability-exploiting attacks had been surveyed in the MPhil thesis [Yoo(2004b)], attack trends of Internet-scale viruses are that they are automatic and sophisticated with intruders misusing infrastructure for their own purposes. To identify Internet-scale viruses, in addition to the usual network control ability of a firewall, data packet detection is compulsory. A packet-header check in a firewall is not sufficient to detect infrastructure based attacks. It is, for example, not sufficient to identify multiple file extension, domain-name-style file names and long subjects in email attachments. Win32/SirCam, Win32/Gibe, Win32/Myparty, Nimda, and Win32/BadTrans misuse this point [Yoo and Ultes-Nitsche(2003), Yoo and Ultes-Nitsche(2004a)].

Anti-virus software can detect viruses in programs using a unique character, called a virus signature, a peculiar attack pattern will also appear in the data packets such as Code Red's packet. IDSs run a process known as anomaly detection. An IDS constantly monitors network traffic and compares the stream of network packets with what it perceives as normal network traffic. Anomaly detection appears to be applicable not only to intrusion detection but also to virus monitoring [Swimmer(2000)], now not being applied to the level of the full network traffic, but to single data packets. The improved virus monitor will examine data packets as usual. Besides checking against known malicious-code patterns, it will check whether it sees a pattern that it perceives as potentially malicious and will react accordingly, e.g. by creating some sort of warnings. However, during the investigation of data packets in a selection of the good packets, a very similar pattern was identified to packets, which seem to contain malicious code, e.g. the BAT911/Chode

worm. These were packets sent by Microsoft Servers to NetBIOS and DNS lookup services. For example, port 137 is reserved for the NetBIOS name service and port 138 is reserved for the NetBIOS datagram service. The subsequent packet was assumed to contain the signature of the "BAT911/Chode" worm even though it was a benign packet (see Table 2.1) [Yoo and Ultes-Nitsche(2002b), InSeon and Ultes-Nitsche(2002)].

TABLE 2.1: Example of NetBIOS name service packets

```
05/24-13:10:13.082716 152.78.70.46:137 -> 152.78.70.127:137
UDP TTL:128 TOS:0x0 ID:47635 IpLen:20 DgmLen:78
Len: 58
.....................................................................
0x0030: 00 00 00 00 00 00 20 45 45 46 44 46 44 45 46 43 .....EEFDFDEFC
0x0040: 41 43 41 43 41 43 41 43 41 43 41 43 41 43 41 43 ACACACACACACACAC
0x0050: 41 43 41 43 41 42 4C 00 00 20 00 01             ACACABL .. ..
```

2.6.2 Dynamic Packet Handling Ability

Although a firewall is able to control a network and maintain its connectivity, it handles packets only statically. Through open ports, a firewall would not inspect/control packets willingly. According to the analysis of distributed denial of service attack tools, it is well known how to use tools such as TFN [Dittrich(1999c)], TFN2K, Trinoo [Dittrich(1999a)] and Stacheldraht [Dittrich(1999b)]. These programs use not only TCP and UDP packets but also ICMP packets. Moreover, because the programs use ICMP_ECHOREPLY packets for communication, it would be very difficult to block attacks without breaking most Internet programs that rely on ICMP. Since TFN, TFN2K and Stacheldraht use ICMP packets, it is much more difficult to detect them in action, and packets will go right through most firewalls. The current only sure way to destroy this channel is to deny all ICMP_ECHO traffic into the network. Furthermore, the tools mentioned above use any port randomly; it is hard to prevent the port from an attack in advance using the fixed port close scheme in current firewalls. Therefore, to prevent degradation of service on the network and to deny this kind of malicious packet, dynamic packet handling on the level of firewalls is crucial [Ultes-Nitsche and Yoo(2003), Yoo and Ultes-Nitsche(2003), Ultes-Nitsche and Yoo(2004), Yoo(2004a)].

2.7 Proposed Firewall Architecture

Janus project intends to extend packet-filter firewalls with intelligent components. The firewall model has packet-based components: a packet verifier, a packet-based classification engine, a smart detection engine, and a policy interpreter. The architectural firewall model is depicted in Figure 2.2. This architectural firewall model has been developed in the MPhil thesis [Yoo(2004b)].

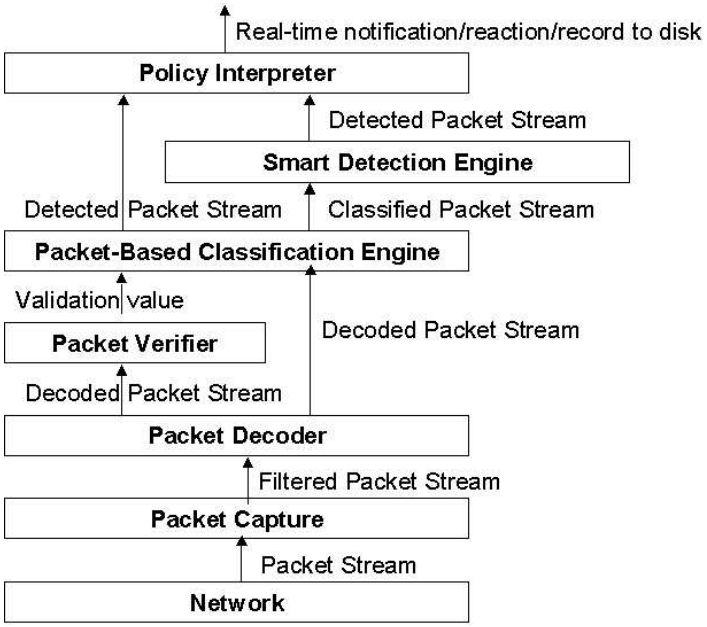

FIGURE 2.2: Packet-Based Detection Components in the intelligent firewall.

Decoded packets pass into the packet verifier and the packet-based classification engine in parallel. The packet verifier checks all protocols' sanity and validates expected usage of protocols. The packet-based classification engine aims to classify packets, which could be malicious and estimate the probability of their maliciousness. On the other hand, the smart detection engine aims at recognizing malicious patterns in data packets that have a certain probability of being malicious. The smart detection engine analyses the payload of the packets and aims to detect anomalous patterns in the payload. Finally, the policy interpreter analyses the information it gets from the two engines and decides on whether to drop the packet or let it pass through the firewall based on its specific security policy [Yoo and Ultes-Nitsche(2003), Ultes-Nitsche and Yoo(2003), Yoo(2004a)].

2.7.1 Packet Verifier

The purposes of the packet verifier are validating compliance to standards, and validating expected usage of protocols e.g. protocol anomaly detection. It aims to cover the TCP/IP/ICMP protocols. The packet verifier checks the protocol header part of packets, verifies packet' size, checks TCP/UDP header length, verifies TCP flags and all packet parameters, does TCP protocol type verification, and analyses TCP protocol header and TCP protocol flags. In the IP protocol, according to the Internet Protocol Standard [Postel(1981b)], an IP header length should always be greater than or equal to the minimal Internet header length (20 octets) and a packet's total length should always be greater than its header length. IP address checks are also important since land attacks use the same IP address for source and destination. According to the TCP standard [Postel(1981c)], neither the source nor the destination TCP port number can be zero, and

TCP flags, e.g. URG and PSH flags, can be only used when a packet carries data. Thus, for instance, combinations of SYN and URG or SYN and PSH become invalid. In addition, any combination of more than one of the SYN, RST, and FIN flags is also invalid. Finally, the packet verifier sends the result of validation to the packet-based classification engine [Yoo and Ultes-Nitsche(2004b), Ultes-Nitsche and Yoo(2004), Yoo(2004c)].

2.7.2 Packet-Based Classification Engine

The purpose of the packet-based classification engine is to make a decision whether the packet classes are filtered into the smart detection engine or are dropped according to their probabilities of being malicious. This classification is based on a structural analysis of data packets. The structural analysis is mainly concerned with information that can be obtained from a packet's header plus certain information in its payload. To make a statistical relation between interesting events among incomplete data, Bayesian networks [Pearl(1988)] or probabilistic graphical models have been chosen, in particular the Naive Bayesian network [Langley and Sage(1994)] among several Bayesian network models. In the MPhil thesis, certain packet characteristics has been analysed that allows me to attach to packets probabilities of their maliciousness. The analysed file characteristics are used as the parameters of the Naive Bayesian network [Yoo and Ultes-Nitsche(2004b), Ultes-Nitsche and Yoo(2004), Yoo(2004c)].

2.7.3 Smart Detection Engine

The smart detection engine deals with the filtered packets, which have a high probability of being malicious, selected from the packet-based classification engine. The smart detection engine aims to learn to distinguish anomalous data packets from normal packets [Cannady and Mahaffey(1998), Lee and Heinbuch(2001)]. However, unlike anti-virus software, this engine does not need to match the infected part of a program exactly [G. Tesauro and Sorkin(1996)]. Detecting known viruses in a system or file is a role of anti-virus software. Note that the smart detection engine deals with virus-infected files rather than file worms. In the file worm case, the packet-based classification engine aims to classify this file worm based on the context information. Currently Self-Organizing Maps (SOMs) [Kohonen(1995)] are applied to the smart detection engine to detect bad patterns. It is aimed to design the SOM in a way that neurons will flag the presence of peculiar patterns in data packets and that the position of the active neurons reflects the position of potentially malicious content in the packet. Basically, all packets with a probability of being malicious above a certain threshold is filtered into the smart detection engine for examination [Yoo(2004d)]. The threshold has to be set in a relatively arbitrary fashion first and then be adapted when fine-tuning is applied to the decision procedures.

2.8 Current Status of Virus Detection

The most popular approach to defend network systems against malicious program is through anti-virus software such as Symantec [Symantec(2002)] and McAfee [McAfee(2002)], as well as server-based scanners that filter email with executable attachments or embedded macros in documents. However, there is still a problem to deal with unknown or variants of viruses.

- Monitoring systems exist through organizations such as WildList [WildList(2001)] and Trend Micro [TrendMicro(2002b)]. WildList is an organization consisting of 64 virus information professionals, who report all computer programs that they have received and positively identified as malicious. This list does not include those cases where an attachment is considered suspicious but not yet classified as malicious, or include any viruses not specifically reported by these 64 participants. This leaves computer systems vulnerable to attack from unreported viral incidents [WildList(2001)]. Since the process of reporting is not automated, malicious programs, especially self-replicating programs, can spread much faster than the warnings generated by WildList.

  Trend depends on a proprietary virus scanner HouseCall [TrendMicro(2002a)], which integrates with the Trend Micro Control Manager to report information about actual virus infections. It attempts to predict virus outbreaks and prevent them pro-actively with the use of a dynamic map to analyse worldwide virus trends in real time [TrendMicro(2002b)]. However, since HouseCall is not widely used, Trend's data is incomplete. Furthermore, if Trend's database is not updated at the time that a virus infects a system, then the virus remains unreported.

- An anti-virus server is defined as a server-side virus-checking program. Its specific name depends on the company that produces it; for example, V3Netscan and V3VirusWall in Ahnlab Inc.[Ahnlab(2002)], and MailMonitor in Sophos [Sophos(2002)]. Anti-virus servers examine network traffic, aiming to prevent malicious code from entering network nodes by detecting known malicious-code patterns, for instance in an email attachment. Apparently, they can detect only known viruses. All of the major anti-virus vendors have produced networked products and systems that scan incoming email. However, because Trojan horses, worms and viruses can spread through local networks, shared hard drives and individual document files, as well as through the Internet, it is always necessary to have virus checking available on each client machine as well as on Internet gateways. Too often, patterns that identify new malware are not ready until days or even weeks after serious damage has been done. New viruses will only become detectable after their pattern characteristics have been analysed and are made available. Looking at techniques applied by other security systems such as intrusion detection systems, seems to benefit virus detection [Swimmer(2000)].

These approaches have been successful in protecting computers against known malicious programs usually employing signature-based methods. Almost all anti-virus products claim that they can detect 100% of known viruses. However, we realize that hundreds of new viruses are created every month, they have not yet provided a means of protecting against unknown viruses, nor do they assist in providing information that may help trace those individuals responsible for creating viruses.

- There have been approaches to detect new or unknown malicious programs by analysing the payload of an attachment. The methods used include heuristics [White(1998)], data mining techniques [Matthew G. Schultz and Zadok(2001b), Matthew G. Schultz and Zadok(2001a)], and neural networks [Kephart(1994)]. However, these methods in general do not perform well enough to detect malicious programs in real time.

  IBM researchers [O.Kephart and C.Arnold(1994)] developed a statistical method for automatically extracting malicious executable signatures. Their research was based on speech recognition algorithms and was shown to perform almost as good as a human expert at detecting known malicious executables. Their algorithm was eventually packaged with IBM's anti-virus software. Lo et al.[R. Lo and Olsson(1995)] presented a method for filtering malicious code based on telltale signs for detecting malicious code. These were manually engineered based on observing the characteristics of malicious code. Unfortunately, a new malicious program may not contain any of the known signatures, so traditional signature-based methods may not detect a new malicious executable. In an attempt to solve this problem, the anti-virus industry generates heuristic classifiers by hand [Gryaznov(1999)]. This process can be even more costly than generating signatures, so finding an automatic method to generate classifiers has been the subject of research in the anti-virus community. To solve this problem, different IBM researchers applied Neural Networks to the problem of detecting boot sector malicious binaries [G. Tesauro and Sorkin(1996)]. A Neural Network is a classifier that aims to explore in human cognition. Because of the limitations of the implementation of their classifier, they were unable to analyse anything other than small boot sector viruses, which comprise about 5% of all malicious binaries. In similar work, William Arnold and Gerald Tesauro [Arnold and Tesauro(2000)] applied the same techniques to Win32 binaries, but because of limitations of the Neural Network classifier, they were unable to have the comparable accuracy when applying to new Win32 binaries.

Chapter 3

Analysis of Vulnerabilities in Network Protocols & Mechanisms

The major difference between a thing that might go wrong
and a thing that cannot possibly go wrong is that
when a thing that cannot possibly go wrong goes wrong,
it usually turns out to be impossible to get at or repair.
- Douglas Adams.

Protocols are created with specifications, known as RFCs, to dictate proper use and communication. An anomaly is defined as something different, abnormal, or not easily classified, or some action, or data that is not considered normal for a given system, or network. Protocol anomaly refers to all exceptions related to protocol format and protocol behaviour with respect to common practice on the Internet and standard specifications. This chapter discusses vulnerabilities in network protocols and mechanisms abused by attacks.

3.1 Protocol Anomaly-Based Attacks

3.1.1 IP Spoofing & Incomplete Three-way Handshake

IP Spoofing is an attack where an intruder pretends to be sending data from its own IP address [Bellovin(1989)]. An IP address either source address or destination address contained in an IP header is the only information needed by an intermediate routing device to make a decision on how to route the IP packet. Anyone who has access to the IP layer can easily modify the source address in the IP header of a packet, spoofing itself as from another host or even from a non-existing host.

Let us also assume that the hosts A and B communicate with one another by following the three-way handshake mechanism of TCP/IP. The handshake method is described below.

A → B: SYN (seq. no. = M)
B → A: SYN (seq. no. = N), ACK (ack. no. = M + 1)
A → B: ACK (ack. no. = N + 1)

Host X does the following to perform IP spoofing. First, it sends a SYN packet to host B with some random sequence number, posing as host A. Host B responds to it by sending a SYN-ACK packet back to host A with an acknowledgment number, which is equal to one, added to the original sequence number. At the same time, host B generates its own sequence number and sends it along with the acknowledgment number. In order to complete the three-way handshake, host X should send an ACK packet back to host B with an acknowledgment number which is equal to one added to the sequence number sent by host B to host A. If we assume that the host X is not present in the same subnet as A or B so that it cannot sniff B's packets, host X has to figure out B's sequence number in order to create the TCP connection. These steps are described below.

X → B: SYN (seq. no. = M), SRC = A
B → A: SYN (seq. no. = N), ACK (ack. no. = M + 1)
X → B: ACK (ack. no. = N + 1), SRC = A

At the same time, host X should take away host A's ability to respond to the packets of host B. To achieve this, X may either wait for host A to go down (for some reason), or block the protocol part of the operating system so that it does not respond back to host B, for example, by flooding B with incomplete connections, such as SYN flooding.

3.1.2 SYN flood attack

In the normal TCP connection establishment, the client system begins by sending a SYN packet to the server. The server then acknowledges the TCP SYN packet by sending SYN-ACK packet to the client. The client then finishes establishing the connection by responding with an ACK message. The connection between the client and the server is then open, and the service-specific data can be exchanged between the client and the server.

The SYN flood attack [CERT(2000a)] exploits the TCP/IP three-way handshake mechanism by having an attacking source host send SYN packets with random source addresses to a victim host. The victim destination host sends a SYN-ACK back to the random source address and adds an entry to the connection queue. Since the SYN-ACK is destined for an incorrect or nonexistent host, the last part of the three-way handshake is never completed, and the entry remains in the connection queue until a timer expires, typically within about one minute. By generating phoney SYN packets from random IP

addresses at a rapid rate, it is possible to fill up the connection queue and deny TCP services to legitimate users.

3.1.3 Ping of Death

Attackers send a fragmented ping request that exceeds the maximum IP packet size (64KB), causing vulnerable systems to crash. The idea behind the ping of death and similar attacks is that the user sends a packet that is malformed in such a way that the target system will not know how to handle the packet. The ping of death attack [CERT(1996)] sent IP packets of a size greater than 65,535 bytes to the target computer. IP packets of this size are abnormal, but applications can be built that are capable of creating them. Carefully programmed operating systems could detect and safely handle abnormal IP packets, but some failed to do this.

3.1.4 Land Attack

The land attack [CISCO(1997)] involves the perpetrator sending a stream of SYN packets that have the source IP address and TCP port number set to the same value as the destination address and port number, i.e., that of the attacked host. Some implementations of TCP/IP cannot handle this theoretically impossible condition, causing the operating system to go into a loop as it tries to resolve repeated connections to itself [Fyodor(1997)].

3.1.5 Smurf attack

The smurf attack [CERT(1998)] is a modification of the ping attack and take advantage of direct broadcast addressing mechanisms by spoofing the target system's IP address and broadcasting ICMP ping requests across multiple subnets. A range of IP addresses from the intermediate system will send pings to the victim, bombarding the victim machine or system with hundreds or thousands of pings. The two main components to the smurf attack are the use of forged ICMP echo request packets and the direction of packets to IP broadcast addresses. On IP networks, a packet can be directed to an individual machine or broadcast to an entire network. In addition, if ICMP echo request packets were directed to IP broadcast addresses from remote locations to generate denial-of-service attacks. Many of the machines on the network will receive this ICMP echo request packet and send an ICMP echo reply packet back. When all the machines on a network respond to this ICMP echo request, the result can be severe network congestion or outages.

3.1.6 Teardrop attack

The teardrop attack [Hoggan(2000)] exploit IP mechanisms involved in the reassembly of packets that have been disassembled for efficient transmission. Packet fragments are deliberately fabricated with overlapping offset fields causing the host to hang or crash when it tries to reassemble them. Under normal conditions, packet fragments will yield a positive integer value as can be derived from the diagram below.

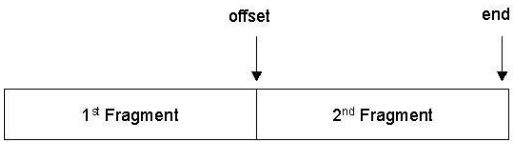

FIGURE 3.1: IP TearDrop Attack - Correct reassemble

However, the teardrop attack sends a fragment that deliberately forces the calculated value for the end pointer to be less than the value for the offset pointer. This can be achieved by ensuring that the second fragment specifies a fragment offset that resides within the data portion of the first fragment and has a length such that the end of the data carried by the second fragment is short enough to fit within the length specified by the first fragment. Diagrammatically this can be shown as follows:

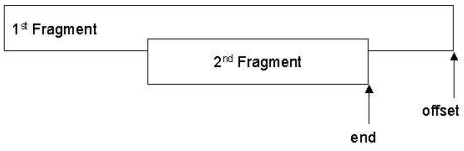

FIGURE 3.2: IP TearDrop Attack - Incorrect reassemble

When the IP module performing the reassembly attempts a memory copy of the fragment data into the buffer assigned to the complete datagram, the calculated length of data to be copied (that is the end pointer minus the offset pointer) yields a negative value. The memory copy function expects an unsigned integer value and so the negative value is viewed as a very large positive integer value. The result of such an action depends upon the IP implementation, but typically causes a stack corruption, failure of the IP module or a system hang.

3.1.7 UDP Flood Attacks

This denial of service attack takes advantage of user datagram protocol (UDP) mechanisms. Since no connection setup is required before data is transferred, it is difficult to bring a host down by flooding the host with just UDP packets. The UDP flood attack [Ferguson and Senie(2000)] uses forged UDP packets to connect the echo service 'echoes' on one machine to the character generation 'chargen' service on the other machine, causing the two machines to consume all available bandwidth on the connection

between them. The echo service of the former machine echoes the data of that packet back to the victim's machine and in turn, the victim's machine responds in the same way. Hence, a constant stream of useless load is created that burdens the network.

3.2 Widespread Malicious Code

A worm is a program that self-propagates across a network exploiting security flaws in widely used services. They are not a new phenomenon, first gaining widespread notice in 1988 [Eichin and Rochlis(1989)]. This project distinguishes between worms and viruses in that the latter require some sort of user action to abet their propagation. As such, viruses tend to propagate more slowly. They also have more mature defences due to the presence of a large anti-virus industry that actively seeks to identify and control their spread. On the other hand, a worm is a computer program which, when it runs, finds other computers that are vulnerable and breaks into them across the network. It then copies itself over, starts itself running on the new hosts, and does the same thing from there. Thus, it can spread exponentially like an epidemic of human disease. The worm has several important aspects [Nazario(2004)]; a spread algorithm for finding other hosts, one or more exploits allowing it to break into other computers remotely, and a payload, which is what it does to your computer after it is broken into it, rather than just using it to spread.

A worm is not the same as a virus. However, they are both malicious code that propagates around the network. There are differences as follows:

- If the mail code can break into another computer and start running there immediately with no human intervention, then it is a worm.
- If the malicious code is carried around in some other content and then may or may not start running on other computers depending on when and whether humans decide to process that content, then it is a virus.

In short, the distinction is made based on whether or not the malicious code is self-activating. By this definition, Code Red, Slammer/Sapphire, and Blaster are worms. ILoveYou and SoBig are viruses. Nimda had both viral and worm spread algorithms. From an operational perspective, the biggest difference is that worms can spread significantly faster, which has strong implications for defences against them. Viruses are more common, however. By and large, existing anti-viral defences are adequate against viruses as long as people deploy and update them properly. However, anti-viral defences are fairly useless against worms, at least during the initial spread of the worm.

This thesis presents an adaptive approach to preventing the damage caused by viruses that travel via email. The approach protects intranet machines from outside infected

machines by spreading email viruses. This directly addresses the two ways that viruses cause damage: fewer machines spreading the virus will reduce the number of machines infected and reduce the traffic generated by the virus. The approach relies on the payload inspection and probabilistic decision about potentially malicious packets.

3.2.1 Activation Techniques

Since most people do not want to have a worm executing on their system, these worms rely on a variety of social engineering techniques. Some worms such as the Melissa virus [CERT(1999a)] indicate urgency on the part of someone you know - "Attached is an important message for you", others, such as the ILoveYou [CERT(2000b)] attack, appeal to individuals' vanity - "Open this message to see who loves you". Although Melissa was a word macro virus - a piece of code written in Microsoft Word's built-in scripting language embedded in a Word document - later human-initiated worms have usually been executable files which, when run, infect the target machine. Furthermore, while some worms required that a user starts running a program, other worms exploited bugs in the software that brought data onto the local system, so that simply viewing the data would start the program running e.g., Klez [Ferrie(2002)].

3.2.2 Propagation

The distribution of code can either be one-to-many, as when a single site provides a worm to other sites, many-to-many, as when multiple copies propagate the malicious code, or a hybrid approach. There are a number of techniques by which a worm can discover new machines to exploit: scanning, external target lists, pre-generated target lists, internal target lists, and passive monitoring. Worms can also use a combination of these strategies.

Two simple forms of scanning are sequential (working through an address block from beginning to end) and random (trying addresses out of a block in a pseudo-random fashion). Due to their simplicity, they are very common propagation strategies, and have been used both in fully autonomous worms [CERT(2001b), eEye(2001)] and worms which require timer or user based activation [MessageLabs(2002)].

3.2.3 Propagation Features of Email Worms

Email worms can be spread via several ways. However, only email propagation in each worm is the interest in this thesis. In addition, the project is looking at *the way of propagation* and *virus mail features* rather than looking for the presence of a virus signature or how a program is infected.

3.2.3.1 W32/Dumaru@MM

The email message created by W32/Gibe [CERT(2002c)] tries to convince users that the attached file is a patch supplied by Microsoft [Authentium(2004)]. In fact, the attached file, patch.exe, is a malicious code. It also uses its own SMTP engine to spread.

3.2.3.2 W32/Myparty

The attached file name of the virus is "www.myparty.yahoo.com." [CERT(2002b)], which cause the default web browser to run unexpectedly. It has a built-in SMTP engine, which it uses to send itself via email to all addresses listed in the infected user's Windows Address Book.

3.2.3.3 VBS/BubbleBoy

This virus, in the same way as Melissa [CERT(1999a)], exploits MS outlook [VirusBulletin(2004)]. However, unlike Melissa, BubbleBoy does not require the user to open a document to run. Just reading an email message is enough to be infected. The virus arrives in what appears to be a standard HTML-enhanced Outlook email message. It copies all the email addresses into the blind carbon copy (BCc) field of a new email message, then sends it.

3.2.3.4 W32/SirCam

The virus appears in an email message written in either English or Spanish with a seemingly random subject line. The email message contains an attachment whose name matches the subject line and has a double file extension (e.g. subject.ZIP.BAT or subject.DOC.EXE). The second extension is .EXE, .COM, .BAT, .PIF, or .LNK [CERT(2001a)]. The attached file contains both the malicious code and the contents of a file copied from an infected system. In addition, this worm includes its own SMTP capabilities, which it uses to propagate via email. It determines its recipient list by recursively searching for email addresses contained in all *.wab (Windows Address Book) files. As a result, propagation via mass emailing causes denial of service conditions.

3.2.3.5 Nimda Worm

This worm propagates through email message consisting of two sections; a blank message, and an executable attachment. The first section is defined as MIME type "text/html", but it contains no text, so the email appears to have no content. The second

section is defined as MIME type “audio/x-wav”, but it contains a base64-encoded attachment file “readme.exe”, which is a binary executable [CERT(2001b)].

3.2.3.6 W32/BadTrans

This malicious Windows program distributes as an email file attachment. The filename in the email attachment of infected email varies from message to message but always has two file extensions such as filename.ext.ext [CERT(2001c)].

Chapter 4

Protocol Anomaly Detection and Verification

Data without generalization is just gossip.
Robert M. Pirsig.

Anomaly in packets is different from packet anomalies, because legitimate packets can contain malicious content, which cause systems to be violated. Therefore, assessing anomaly in packets is not only to deal with anomalous packets, but also to examine legitimate packets, which contain malicious content. However, important to note is that not all threats or attacks exhibit themselves as protocol anomalies. Some types of application logic attacks, denial of service attacks, viruses, and reconnaissance methods [1] [Chmielarski(2001)] all appear as perfectly legitimate network traffic. On the other hand, there are some odd-looking but legitimate traffic includes [Julia Allen(2000)], for instance, storms of FIN and RST packets, fragmented packets with the "don't fragment" flag set, legitimate tiny fragments, and data that is different when retransmitted. For this reason, a well-built detection system will rely on multiple detection mechanisms, each covering some portion of the threat space. This is often referred to as 'defence in depth'. In this chapter, requirements of network protocols and a TCP verification model are presented. Then a protocol verification program, *Packet Verifier* is introduced using SDL model.

[1] Generally scanning methods, the most popular reconnaissance methods, besides general scanning, was DNS version query, followed by queries to RPC services.

4.1 Requirements of Network Protocols For Anomaly Detection

This section discusses requirements of network protocols through RFCs, several TCP/IP books such as [Stevens(1994), Stevens and Wright(1995)], and research papers for IDS traffic analysis like [M. Handley and Paxson(2001), Shankar and Paxson(2003)]. Internet protocol specifications do not always accurately specify the complete behaviour of protocols, in particular for rare or exceptional conditions. In addition, different operating systems and applications implement different subsets of the protocols.

4.1.1 IP Protocol

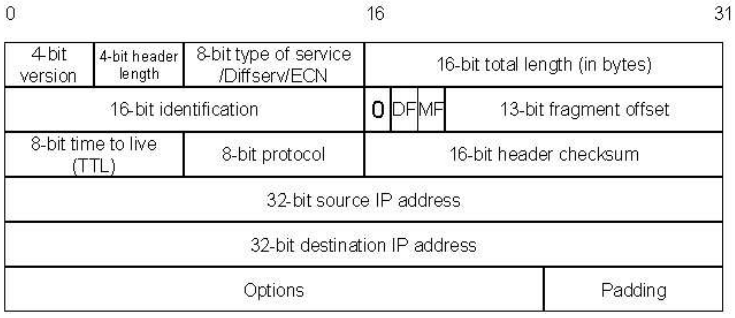

FIGURE 4.1: IP v4 header.

The requirements of the IP header part are based on the Internet protocol (IP) specification [Postel(1981b)], other relative RFCs in each related field and other IDS traffic analysis work like [M. Handley and Paxson(2001)]. Note that the maximum IP packet size is 64 Kbytes. The IPv4 header is depicted in Figure 4.1.

- **Header length:** If the header length field is less than 20 bytes (the header is incomplete), or, if the header length field exceeds the packet length, the packet should be discarded. Note that if the header length is greater than 20 bytes, this indicates options are present.
- **Type Of Service/Diffserv [Grossman(2002)]/ECN (Explicit Congestion Notification) [Ramakrishnan and Floyd(1999)] :** These bits have been reassigned to differentiated services [K. Nichols and Black(1998)] and explicit congestion notification [Ramakrishnan and Floyd(1999)]. If a site does not actually use Diffserv mechanisms for incoming traffic, the bits should be zero. If these bits are not being used internally, the bits should be zero.
- **Total length:** Total length must contain the total length of the IP datagram. This includes IP header, e.g., ICMP or TCP or UDP header and payload size in

bytes. If the total length field does not match the actual total length of the packet as indicated by the link layer, then packets whose length field exceeds their link layer should be discarded.

- **IP Identification number:** This field uniquely identifies each datagram sent by a host. It normally increments by one each time a datagram is sent. It is used to distinguish the fragments of one datagram from those of another. The identification field, the fragment offset field and the total length field provide sufficient information to reassemble datagrams. RFC 791 [Postel(1981b)] states that fragmentation is necessary when it originates in a local network that allows a large packet size and must traverse a local network that limits packets to a smaller size to reach its destination.

- **Must be zero:** The current IP specification (RFC 791 [Postel(1981b)]) states that the bit between IP identifier and DF must be zero. Hence, the packet with a non-zero set bit should be discarded.

- **Don't Fragment(DF) flag:** An Internet datagram can be marked "Don't Fragment". Any Internet datagram so marked is not to be fragmented under any circumstances. If Internet datagram marked "Don't Fragment" cannot be delivered to its destination without fragmenting it, the packets should be discarded. If DF is set, and the Maximum Transmission Unit (MTU) anywhere in the internal network is smaller than the MTU on outside network to the site, DF on incoming packets should be zero. If the packet with the "Don't Fragment (DF)" bit set in the IP header, is too large for a router to forward on to a particular link, the router must send an "ICMP Destination Unreachable - Fragmentation needed" message to the source address [Lahey(2000)]. The MTU of a given network link specifies the largest allowable size of an IP packet on that link. Packets arriving with DF set and a non-zero fragmentation offset are illegal. Hence, such packets should be discarded.

- **More Fragments (MF) flag, Fragment Offset:** These two fields are treated together because they are interpreted together for IP fragmentation. Packets where the length plus the fragmentation offset exceeds 65535 are illegal. Hence, the packets should be discarded.

- **Time to live (TTL):** As with DF, an attacker can use TTL to manipulate the packet. Therefore, it is necessary to restore a TTL if that is larger than the longest path across the internal site. If packets arrive that have a TTL lower than the configured minimum, then it is necessary to restore the TTL to the minimum. Since TTL spoofing is considered nearly impossible, a mechanism based on an expected TTL value can provide a simple and reasonably robust defence from infrastructure attacks based on forged protocol packets [V. Gill and Meyer(2004)]. Note that TTL should be the same for all fragments of a given IP packet.

- **Source address:** If the source address of an IP packet is invalid in some way, for example, 127.0.0.1 (localhost), 0.0.0.0 and 255.255.255.255 (broadcast), multicast (class D) and class E address, source address is the same as destination address (abused by the land attacks [CISCO(1997)]) , the packet should be discarded.
- **Destination address:** Like source address, if invalid destination address occurs with local broadcast address (abused by the smurf attacks [CERT(1998)]), localhost, broadcast address, and class E address, which are currently unused, the packet should be discarded.
- **IP options:** IP packets may contain IP options that modify the behaviour of internal hosts, or cause packets to be interpreted differently. Therefore, remove IP options from incoming packets.
- **Padding:** The receiver explicitly ignores the padding field at the end of a list of IP options, so it is safe to zero the padding bytes.

The protocol field indicates the next-layer protocol, such as TCP or UDP. According to local site policy, a firewall can block traffic based on it. Moreover, regarding IP header checksums, routers normally discard packets with incorrect IP checksums.

4.1.1.1 IP Fragmentation

Fragmentation is necessary in order for traffic, which is being sent across different types of network media to arrive successfully at its intended destination. The reason for this is that different types of network media and protocols have different rules involving the maximum size allowed for datagrams on its network segment or MTU [Anderson(2001a)]. Whenever the IP layer receives an IP datagram to send, it determines which local interface the datagram is being sent on (routing), and queries that interface to obtain its MTU. IP compares the MTU with the datagram size and performs fragmentation, if necessary. Fragmentation can take place either at the original sending host or at an intermediate router [Stevens(1994)]. In order for a fragmented packet to be successfully reassembled at the receiver of the fragments, the receiver should use these checks in RFC 791 [Postel(1981b)]. All of this information will be contained in the IP header.

- **IP identification number field.** Must share a common IP identification number to ensure that fragments of different datagrams are not mixed.
- **IP fragment offset field.** The fragment offset field tells the receiver the position of a fragment in the original datagram.
- **IP total length field.** Each fragment tells the length of the data carried in the fragment. The fragment offset and length determine the portion of the original datagram covered by this fragment.

- **IP more fragmentation (MF) field.** The fragment must tell whether more fragments follow this one or not using DF field.

RFC 791 [Postel(1981b)] states that every Internet module must be able to forward a datagram of 68 octets without further fragmentation. This is because an Internet header may be up to 60 octets, and the minimum fragment is 8 octets. However, for the purpose of security, it is not sufficient to merely guarantee that a fragment contains at least 8 octets of data beyond the IP header. Thus, RFC 1858 [G. Ziemba and Traina(1995)] states also that if the router's filtering module enforces a minimum fragment offset for fragments that have non-zero offsets, it can prevent overlaps in filter parameter regions of the transport headers. In the case of TCP, the TCP flags field is never contained in a non-zero-offset fragment. If a TCP fragment has fragment offset set with non-zero (e.g. FO=1), it should be discarded because it starts only eight octets into the transport header.

4.1.2 ICMP Protocol

IP itself has no mechanism for establishing and maintaining a connection, or even containing data as a direct payload. The Internet control messaging protocol (ICMP) [Postel(1981a)] is merely an addition to IP to carry error, routing and control messages and data. ICMP is used to handle errors and exchange control messages. ICMP can also be used to determine if a machine on the Internet is responding. To do this, an ICMP echo request packet is sent to a machine. If a machine receives that packet, that machine will return an ICMP echo reply packet. ICMP is used to convey status and error information including notification of network congestion and of other network transport problems. ICMP can also be a valuable tool in diagnosing host or network problems. Note that total ICMP header length is 8 bytes and the maximum requested data echo size is 548 bytes.

- **ICMP type:** The message type, for example 0 is echo reply, 8 is echo request, 3 is destination unreachable.
- **ICMP code:** This is significant when sending an error message (unreachable), and specifies the kind of error.
- **ICMP checksum:** The checksum for the ICMP header, and it is the same value as the IP checksum.
- **ICMP id:** It is used in echo request/reply messages, to identify the request.
- **ICMP sequence:** It identifies the sequence of echo messages, if more than one is sent.

4.1.3 UDP Protocol

The user datagram protocol (UDP) [Postel(1980)] is a transport protocol for sessions that need to exchange data. Both transport protocols, UDP and TCP provide 65535 different source and destination ports. The destination port is used to connect to a specific service on that port. The UDP header is depicted in Figure 4.2.

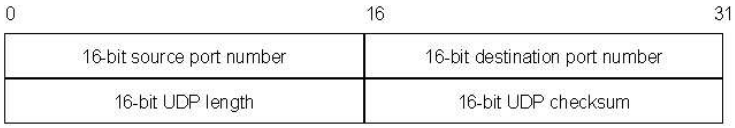

FIGURE 4.2: UDP header.

- **Source port number:** It is the source port that a client binds to, and the contacted server will reply back to in order to direct its responses to the client.
- **Destination port number:** It is the destination port, which a specific server can be contacted on.
- **Length:** It is the length of UDP header and payload data in bytes. If it does not match length as indicated by IP total length, then the packet should be discarded.
- **Checksum:** It is the checksum of header and data.

4.1.4 TCP Protocol

The transmission control protocol (TCP) [Postel(1981c)] is the most used transport protocol that provides mechanisms to establish a reliable connection with some basic authentication, using connection states and sequence numbers. The TCP header is illustrated in Figure 4.3. The size of the TCP header is 20 bytes, without counting its options. Each TCP segment contains the source and destination port number to identify the sending and receiving of application programs, respectively. The sequence number is essential to maintain the bytes of data from the sender to the receiver in proper order. By communicating the sequence number and the corresponding acknowledgment number, the sender and the receiver can determine lost or retransmitted data in the connection. There are six flag bits in the TCP header, namely URG, ACK, PSH, SYN, and FIN. At any given time, one or more of these flag bits can be set.

TCP provides flow control by advertising the window size. The checksum covers TCP header and TCP data and assists in determining any error in transmission of TCP header or data. TCP's urgent mode is a method for the sender to transmit emergency/urgent data. The urgent pointer is valid only if the URG flag is set in the header. It helps to locate the sequence number of the last byte of urgent data. There is an optional options field as well, taking care of vendor specific information.

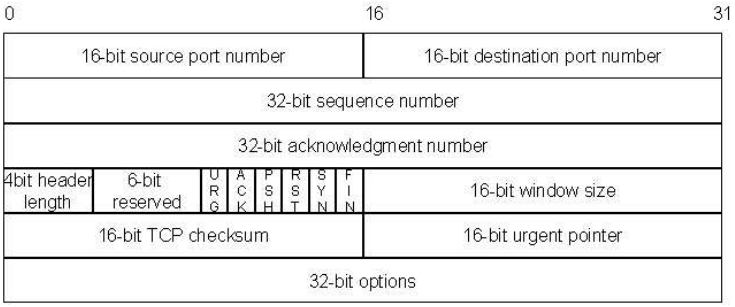

FIGURE 4.3: TCP header.

- **Source and destination port number:** The source port and the destination port. Ports are used by the kernel to identify network processes. Together with an IP address, a TCP port provides an endpoint for network communication. Note that the source address and the port should not be the same as the destination address and the port (abused by the land attack [CISCO(1997)]).

- **Sequence number:** The sequence number is used to enumerate the TCP segments. The data in a TCP connection can be contained in any amount of segments (single TCP datagrams), which will be put in order and acknowledged. TCP acknowledges all data bytes received from the other end. The initial sequence number should be chosen randomly or the sequence number incremented randomly. Bellovin [Bellovin(1989)] describes a fix for TCP that involves partitioning the sequence number space. Each connection would have its own separate sequence number space.

- **Acknowledgment number:** Every packet that is sent and a valid part of a connection is acknowledged with an empty TCP segment with the ACK flag set and this acknowledge field containing the value of the next expected sequence number from the other side and acknowledges all data from the other side up through this acknowledgment number minus one.

- **Header length:** The header length gives the length of the header in 32-bit words. This is required because the length of the options field is variable. With a 4-bit field, TCP is limited to a 60-byte header. Without options, the normal value of this field is 5 (20 bytes) (e.g., 5 in decimal and 0101 in binary), thus if it is less than 5 or if it is beyond end of packet, then the packet should be discarded.

- **Reserved:** The current TCP specification [Postel(1981c)] states that this field must be zero.

- **Flags:** This field consists of six binary flags.
(1) URG: Urgent. It implies the urgent pointer is valid. Segment will be routed faster, used for termination of a connection or to stop processes (using the telnet protocol). This is TCP's way of implementing out of band data. It can be used

only when a packet carries data. If an URG packet without an ACK flag, the packet should be discarded.
(2) ACK: Acknowledgement. It is used to acknowledge data and in the second and third stage of a TCP connection initiation.
(3) PSH: Push. The systems IP stack will not queue this data, but rather pass it to the application as soon as possible. This flag should always be set in interactive connections such as telnet and rlogin. It can be used only when a packet carries data. If a PSH packet without an ACK flag, the packet should be discarded.
(4) RST: Reset. It tells the peer that the connection has been terminated. All memory structures are torn down. If a RST packet comes in, data should be removed.
(5) SYN: Synchronization. It means the synchronize sequence number field is valid. A segment with the SYN flag set indicates that a client wants to initiate a new connection to the destination port. This flag is only valid during the three-way handshake. Note that combination of SYN and URG, or SYN and PSH is invalid. If a SYN packet with a RST flag, the packet should be discarded.
(6) FIN: Final. The connection should be closed, the peer is supposed to answer with one last segment with the FIN flag set as well. If a FIN packet without an ACK or SYN flag, the packet should be discarded without responding.

- **Window size:** The amount of bytes that can be sent before the data must be acknowledged with an ACK before sending more segments.

- **TCP checksum:** It is the checksum of pseudo header, TCP header and payload. The pseudo is a structure containing IP source and destination address, 1 byte set to zero, the protocol (1 byte with a decimal value of 6), and 2 bytes (unsigned short) containing the total length of the TCP segment. If it is incorrect, the packet should be discarded.

- **Urgent pointer:** Only used if the urgent flag is set, otherwise this flag should be zero. It points to the end of the payload data that should be sent with priority.

- **Options:**
  (1) MSS (Maximum Segment Size) option [Lahey(2000)]: The TCP MSS (Maximum Segment Size) [2] option is only allowed in SYN packets. A TCP packet with an MSS option and without a SYN flag is illegal. The MSS advertised at the start of a connection should be based on the MTU of the interfaces on the system.
  (2) WS (Window Scale) option [V. Jacobson and Borman(1992)]: The three-byte window scale option may be sent in a SYN segment by TCP. It has two purposes: (1) indicate that the TCP is prepared to do both send and receive window scaling, and (2) communicate a scale factor to be applied to its receive window. Thus, a

[2] The maximum amount of TCP data that a node can send in one segment. This should be the size of the receiver's reassembly buffer to try to avoid fragmentation. The equivalent at the physical layer is "Maximum Transmission Unit (MTU)".

TCP that is prepared to scale windows should send the option, even if its own scale factor is 1. The scale factor is limited to a power of two and encoded logarithmically, so it may be implemented by binary shift operations. This option is an offer, not a promise; both sides must send window scale options in their SYN segments to enable window scaling in either direction. This option may be sent in an initial SYN segment (i.e., a segment with the SYN bit on and the ACK bit off). It may also be sent in a SYN-ACK segment, but only if a Window scale option was received in the initial SYN segment. A window scale option in a segment without a SYN bit should be ignored. The window field in a SYN (i.e., a SYN or SYN-ACK) segment itself is never scaled. If any packet does not have a SYN flag set on, the option should be removed.

(3) SACK (Selective Acknowledgement) option [M. Mathis and Romanow(1996)]: With selective acknowledgments, the data receiver can inform the sender about all segments that have arrived successfully, so the sender need retransmit only the segments that have actually been lost. The selective acknowledgment extension uses two TCP options. The first is an enabling option, "SACK-permitted", which may be sent in a SYN segment to indicate that the SACK option can be used once the connection is established. The other is the SACK option itself, which may be sent over an established connection once permission has been given by SACK-permitted. The Sack-Permitted option must not be sent on non-SYN segments. If the data receiver has not received a SACK-Permitted option for a given connection, it must not send SACK options on that connection.

(4) T/TCP option [Braden(1995)]: T/TCP is a small set of extensions to make a faster, more efficient TCP. It is designed to be a completely backward compatible set of extensions to speed up TCP connections. T/TCP achieves its speed increase from two major enhancements over TCP: TAO and TIME_WAIT state truncation. TAO is TCP Accelerated Open, which introduces new extended options to bypass the three-way handshake entirely. Using TAO, a given T/TCP connection can approximate a UDP connection in terms of speed, while still maintaining the reliability of a TCP connection. In most single data packet exchanges (such is the case with transaction-oriented connections like HTTP), the packet count is reduced by a third. The second speed up is TIME_WAIT state truncation. TIME_WAIT state truncation allows a T/TCP client to shorten the TIME_WAIT state. This can allow a client to make more efficient that use of network socket primitives and system memory. This is an experimental TCP extension for efficient transaction-oriented (request/response) service. It is safe to remove this option.

(5) TS (TimeStamps) option [Stevens(1994)]: TCP is a symmetric protocol, allowing data to be sent at any time in either direction, and therefore timestamp echoing may occur in either direction. For simplicity and symmetry, we specify that timestamps always be sent and echoed in both directions. For efficiency, we combine the timestamp and timestamp reply fields into a single TCP Timestamps Option. The

Timestamps option carries two four-byte timestamp fields. The Timestamp Value field (TSval) contains the current value of the timestamp clock of the TCP sending the option. The Timestamp Echo Reply field (TSecr) is only valid if the ACK bit is set in the TCP header; if it is valid, it echoes a timestamp value that was sent by the remote TCP in the TSval field of a Timestamps option. When TSecr is not valid, its value must be zero. The TSecr value will generally be from the most recent Timestamp option that was received. A TCP may send the Timestamps option (TSopt) in an initial SYN segment (i.e., segment containing a SYN bit and no ACK bit), and may send a TSopt in other segments only if it received a TSopt in the initial SYN segment for the connection. If it is not negotiated in SYN, it is safe to remove it.
(6) MD5 Signature Option [Heffernan(1998)]: The security of this option relies heavily on the quality of the keying material used to compute the MD5 signature [Leech(2003)]. If MD5 is used in SYN, then non-SYN packets without it should be discarded.
(7) Other options: It is safe to remove other options.

4.2 TCP Runtime Verification Model

4.2.1 Current TCP Model

The transmission control protocol (TCP) [Postel(1981c)] is the most common transport layer protocol used in modern networking environments. TCP provides reliable data transfer between different application processes over the network. TCP provides flow control and congestion control [M. Handley and Floyd(2000)] as well. Initiation, establishment, and termination of a connection are governed by the TCP state-transition diagram, which consists of well-defined states and transition arcs between these states (see Figure 4.4.). Nevertheless, during the past two decades, many security problems of the TCP/IP protocol suite have been discovered [Bellovin(1989)]. Meanwhile, the network hackers created a large number of intrusion methods to exploit those vulnerabilities.

The TCP state-transition diagram (see Figure 4.4.) is very closely associated with timers. There are various timers associated with connection establishment or termination, flow control, and retransmission of data. A connection-establishment timer is set when the SYN packet is sent during the connection-establishment phase. If a response is not received within 75 seconds (in most TCP implementations), the connection establishment is aborted. A FIN_WAIT_2 timer is set to 10 minutes when a connection moves from the FIN_WAIT_1 state to the FIN_WAIT_2 state [Stevens and Wright(1995)]. If the connection does not receive a TCP packet with the FIN bit set within the stipu-

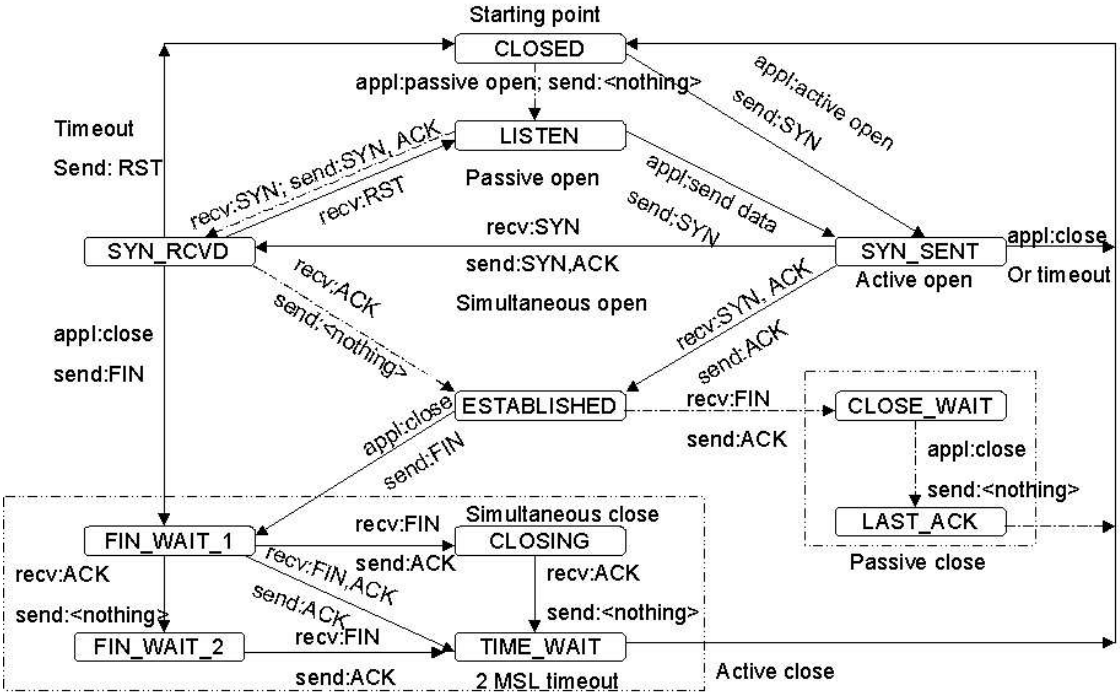

FIGURE 4.4: TCP connection state diagram from TCP/IP Illustrated Vol. 1 - The Protocols, 18.6

lated time, the timer expires. If no FIN packet arrives within this time, the connection is dropped. There is a TIME_WAIT timer, often called a 2 MSL (Maximum Segment Lifetime) [3] timer. It is set when a connection enters the TIME_WAIT state. When the timer expires, the kernel data-blocks related to that particular connection are deleted, and the connection is terminated. A keepalive timer can be set which periodically checks whether the other end of the connection is still active. If the SO_KEEPALIVE socket option is set, and if the TCP state is either ESTABLISHED or CLOSE_WAIT, and the connection is idle, the probes are sent to the other end of the connection once every two hours. If the other end does not respond to a fixed number of these probes, the connection is terminated.

4.2.1.1 Problems with Extraneous State Transitions

Let us consider a sequence of packets between hosts X and A. Intruder-controlled host X needs to perform the following steps to wedge A's operating steps so that it cannot respond to unexpected SYN-ACKs from other hosts for as long as two hours.

1. Host X sends a packet to host A with SYN and FIN flags set. Host A responds with an ACK packet. Host A changes its state from LISTEN to SYN_RCVD, and then to CLOSE_WAIT.

[3] Common implementation values for 2 MSL are 1 minute or 2 minutes.

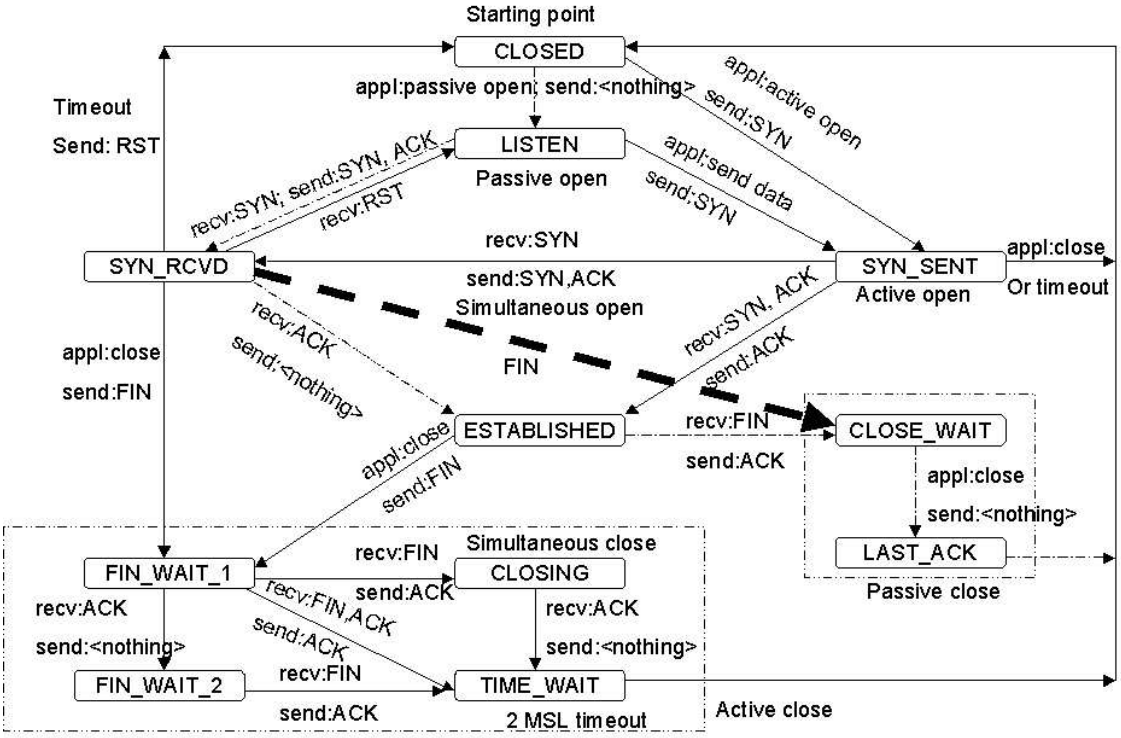

FIGURE 4.5: Extraneous state in the TCP state-transition diagram [Extention of Figure 4.4]

2. Host X does not send any more packet to host A, thus preventing any TCP state-transitions in host A.

Examining the state-transition diagram in Figure 4.5, we observe that host A is initially in state LISTEN. When it receives the packet from host X, it starts processing the packet. It processes the SYN flag first, then transitions to the SYN_RCVD state. Then it processes the FIN flag and performs a transition to the state CLOSE_WAIT. Had the previous state been ESTABLISHED, this transition to the CLOSE_WAIT state would have been a normal transition. However, a transition from SYN_RCVD state to the CLOSE_WAIT state is not defined in the TCP specification. This phenomenon occurs in several TCP implementations, such as those in the operating systems SUNOS 4.1.3, SVR4, and ULTRIX 4.3 [Gupta and Mukherjee(1996)]. Thus, contrary to specification, there exists in several TCP implementations a transition arc from the state SYN_RCVD to the state CLOSE_WAIT, as shown in Figure 4.5.

In this attack scenario, the TCP connection is not yet fully established since the three-way handshake is not completed; thus, the corresponding network application never got the connection from the kernel. However, host A's TCP machine is in CLOSE_WAIT state and is expecting the application to send a close signal so that it can send a FIN packet to host X and terminate the connection. This half-open connection remains in the socket-listen queue and the application does not send any message to help TCP perform

any state-transition. Thus, host A's TCP machine stuck in the CLOSE_WAIT state. If the keep-alive timer feature is enabled, TCP will be able to reset the connection and perform a transition to the CLOSED state after a period of usually two hours. Thus, we observe that extraneous state-transitions exist in several implementations of TCP and these may lead to severe security violations of the system.

4.2.1.2 Problems with Simultaneous Open

Let us consider the sequence of steps followed by an intruder-controlled host X and host A. This attack scenario was mentioned in [Gupta and Mukherjee(1996)]. Through these steps, host X is successfully able to stall a port of host A.

1. Host X sends a SYN packet to host A. A TCP connection is established between hosts X and A. Host A sends a SYN packet to host X in order to start a TCP connection and performs a state-transition to the state SYN_SENT.
2. When host X receives the SYN packet from host A, it sends a SYN packet back in response.
3. When host A receives this packet, it assumes that a simultaneous open connection is in progress; it sends out a SYN-ACK packet to host X and at the same time switches off the connection-establishment timer and makes a state-transition to state SYN_RCVD.
4. Host X receives the SYN-ACK packet from host A but does not send back any packet.
5. Host A is expecting a SYN-ACK from the host X. Since host X does not send back any packet, host A is stalled in the state SYN_RCVD.

4.2.2 Generating TCP Verification Model

4.2.2.1 Removing Unnecessary States In Implementation

In practical internetworking with TCP, for example, if an application prematurely closes in SYN_RCVD state, the specification requires the implementation to close the connection using the FIN, however, the transition from SYN_RECV to FIN_WAIT_1 cannot possibly happen (see Figure 4.6). These transactions are part of ideal connection terminations, but are not applicable to real TCP implementations. The working Linux source (version 2.4.19) does not support this; the Linux implementation dropped the connection and there was a comment that acknowledges the incorrect handling of this case. Furthermore, some implementation fails to response a RST packet in SYN_RCVD

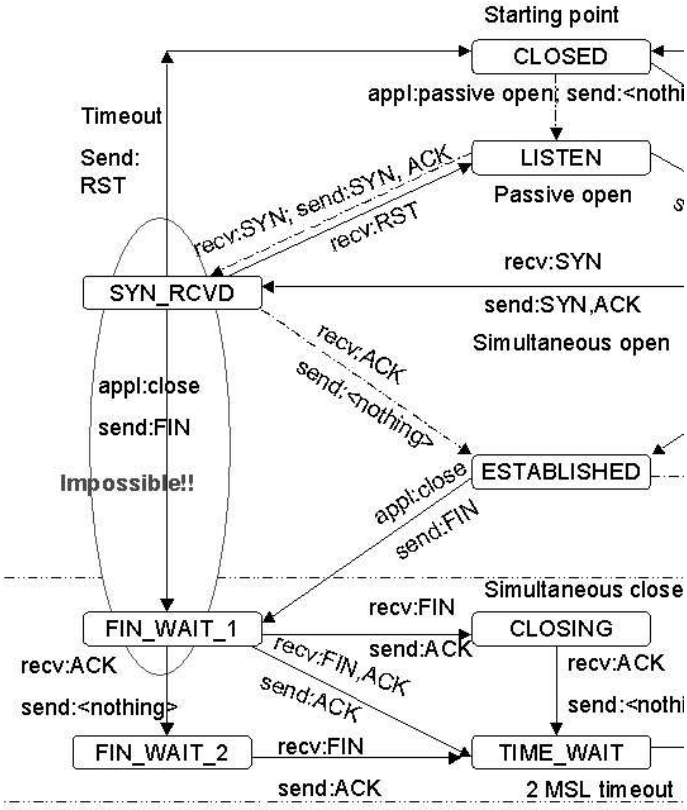

FIGURE 4.6: Impossible state transition in the existing TCP Protocol State Machine

state, which cause prematurely exit before processing the RST flag. Besides, there are more unsupported states. The following description of TCP State Transition Diagram is excerpted from TCP/IP Illustrated Vol. 1 - The Protocols, 18.6 [[Stevens(1994)]]:" The state transition from LISTEN to SYN_SENT is legal but is not supported in some implementations, e.g., Berkeley-derived implementations." In FIN_WAIT_2 state, "we have sent our FIN and the other end has acknowledged it. Unless we have done a half-close [4], we are waiting for the application on the other end to recognize that it has received an end-of-file notification and close its end of the connection, which sends us a FIN. Only when the process at the other end does this close will our end move from the FIN_WAIT_2 to the TIME_WAIT state. This means our end of the connection can remain in this state forever. The other end is still in the CLOSE_WAIT state, and can remain there forever, until the application decides to issue its close. Many Berkeley-derived implementations prevent this infinite wait in the FIN_WAIT_2 state as follows. If the application that does the active close does a complete close, not a half-close indicating that it expects to receive data, then a timer is set. If the connection is idle for 10 minutes plus 75 seconds, TCP moves the connection into the CLOSED state. A comment in the code acknowledges that this implementation feature violates the protocol

[4]TCP provides the ability for one end of a connection to terminate its output, while still receiving data from the other end. This is called a half-close. The socket API supports the half-close, if the application calls shutdown with a second argument of 1, instead of calling close. Most applications, however, terminate both directions of the connection by calling close.

specification."

4.2.2.2 Reorganizing Sequences Of States

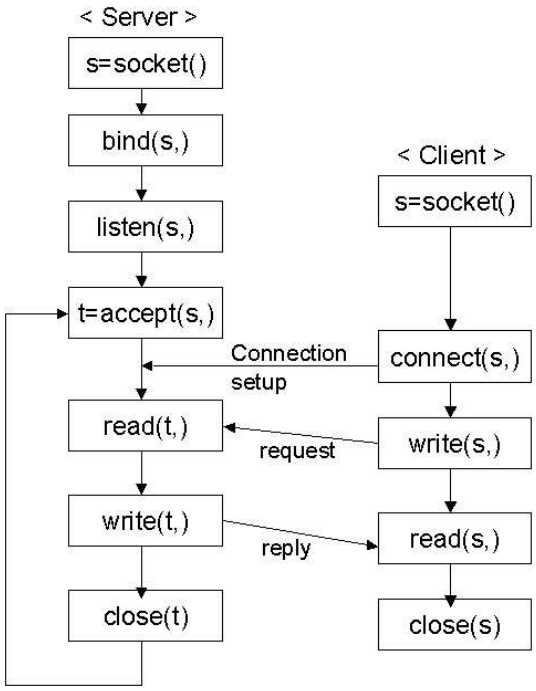

FIGURE 4.7: Client/Server Interaction

Client/server interaction is illustrated in Figure 4.7. To generate a TCP verification model, client/server packet interaction is observed neither from the server nor from the client side but from the viewpoint of an independent observer. For example, in firewalls' view, close observation of packet communication from external clients to inside servers or clients is worthwhile. Since a firewall is neither a server nor a client, it needs to look at the packet-transition sequence of both sides. The firewall also can take communication flags accordance with states into account.

Figure 4.8 illustrates a TCP verification model after eliminating unnecessary states in implementation. This TCP verification model captures only the essential details of the TCP protocol and accepts a superset of what is permitted by the standards. It is still sufficient to deal with incomplete protocol runs meeting the standards (such as abuse of incomplete three-way handshake). In order to achieve the goal of identifying all necessary TCP transition for a set of reachable states, first it is necessary to search through all the reachable states from the initial state to find their necessary transitions. It then reduces the number of the searched states.

A TCP connection is always initiated with the three-way handshake, which establishes and negotiates the actual connection over which data will be sent. The whole session begins with a SYN packet, then a SYN-ACK packet and finally an ACK packet to

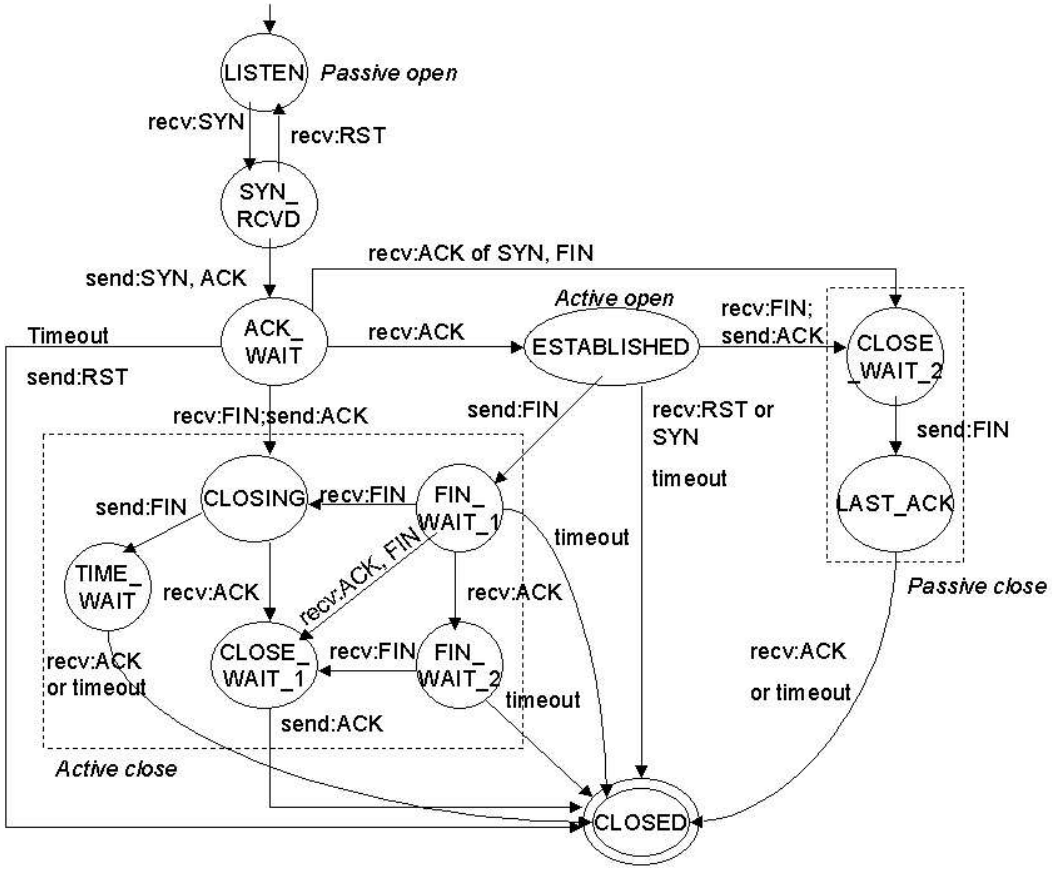

FIGURE 4.8: TCP Verification Model (First)

acknowledge the whole session establishment. In Figure 4.8, a new session starts in the LISTEN state. Data transfer takes place in the connection ESTABLISHED state. If the TCP connection is initiated, then the state machine goes through SYN_RCVD and ACK_WAIT states to reach the ESTABLISHED state. In order to tear down the connection, either side can send a TCP segment with the FIN bit set. If an internal host sends the FIN packet, the state machine waits for an ACK of FIN to come in from an external host. This scenario is represented by the states FIN_WAIT_1, FIN_WAIT_2, CLOSING and CLOSE_WAIT_1 states. It is also possible that an external host send a FIN packet after the ESTABLISHED state. In this case, we may receive a FIN, or a FIN-ACK from the external host. This scenario is represented by the CLOSE_WAIT_2 and LAST_ACK states.

4.2.2.3 Removing Server-side-dependent Termination

In view of an independent observer like firewalls, the model can become even simpler and thus much easier to apply to a real system. We do not need to look at which state occurs on the server side where the monitoring firewall is located, since server side initiates termination the independent observer does not need to verify next states after termination process. Besides, to make the verification steps more abstract, we can

ignore three parts, which are marked by a circle as shown in Figure 4.9.

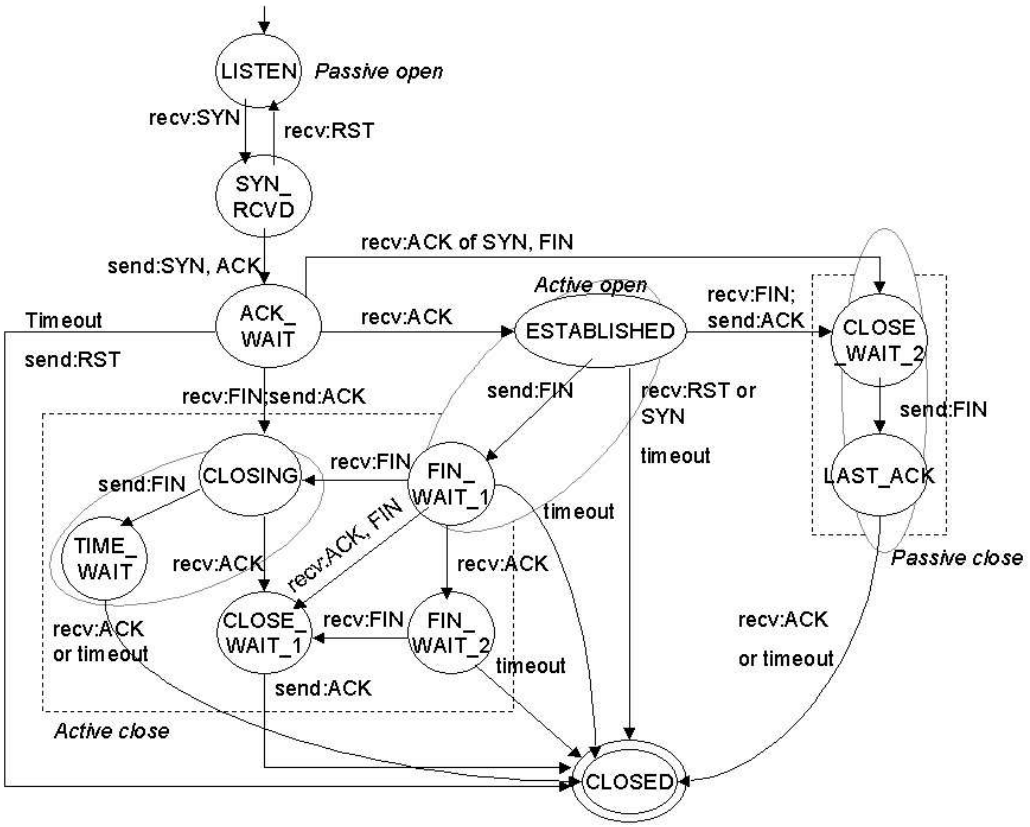

FIGURE 4.9: Three Groups in TCP Verification Model (Second)

Some terms are used; an internal host as server side, an external host as client side to make sure that the independent observer like firewalls is located in server side to monitor all packets.

1. ESTABLISHED to FIN_WAIT_1 state. This state transition requires request of an internal host's requests in order to tear down the connection. In particular, since the request comes from the internal host, the independent observer does not need to trace it down.

2. CLOSING to TIME_WAIT state. Since state CLOSING is reached after a FIN has occurred, Janus in this state can ignore other packets from the client and simply drop them in any case.

3. CLOSE_WAIT_2 to LAST_ACK state. This represents the normal closing state of an external client's request after a connection has been established. The independent observer does not need to trace down the server-side state change.

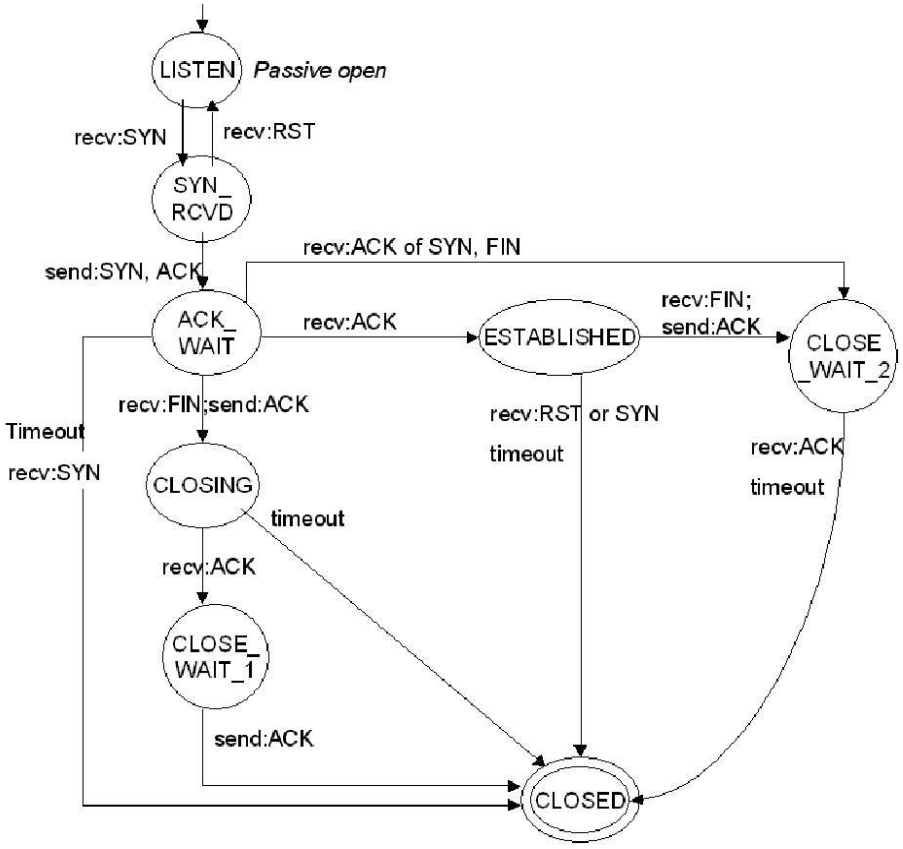

FIGURE 4.10: Simplified TCP Verification Model (Final)

TABLE 4.1: TCP states table

Current State	New State	Action
LISTEN	SYN_RCVD	recv: SYN
SYN_RCVD	LISTEN	recv: RST
SYN_RCVD	ACK_WAIT	send: SYN, ACK
ACK_WAIT	ESTABLISHED	recv: ACK
ACK_WAIT	CLOSE_WAIT_2	recv: ACK of SYN, FIN
ACK_WAIT	CLOSED	timeout (60 sec) or recv: SYN
ACK_WAIT	CLOSING	recv: FIN, send: ACK
ESTABLISHED	CLOSE_WAIT_2	recv: FIN, send: ACK
ESTABLISHED	CLOSED	recv: RST or SYN, or timeout (30 min)
CLOSING	CLOSED	timeout (10 sec)
CLOSING	CLOSE_WAIT_2	recv: ACK
CLOSE_WAIT_2	CLOSED	recv: ACK, or timeout (10 sec)

4.2.2.4 Simplified TCP Verification Model

After leaving out the three cases above, it is not necessary for ACK_WAIT state to send RST to client side, but need to check whether server side receives SYN or not.

The simplified TCP verification model is depicted in Figure 4.10 and the associated TCP state table with action is presented in Table 4.1 to monitor and control TCP state transitions. They may change according to the security policy, however, the default values should be fairly well established in practice. To reduce clutter, the following classes of abnormal transitions are not shown: conditions where an abnormal packet is discarded without a state transition, e.g., packets received without correct sequence numbers after connection establishment and packets with incorrect flag settings.

4.2.2.5 Example Cases of TCP State Transition

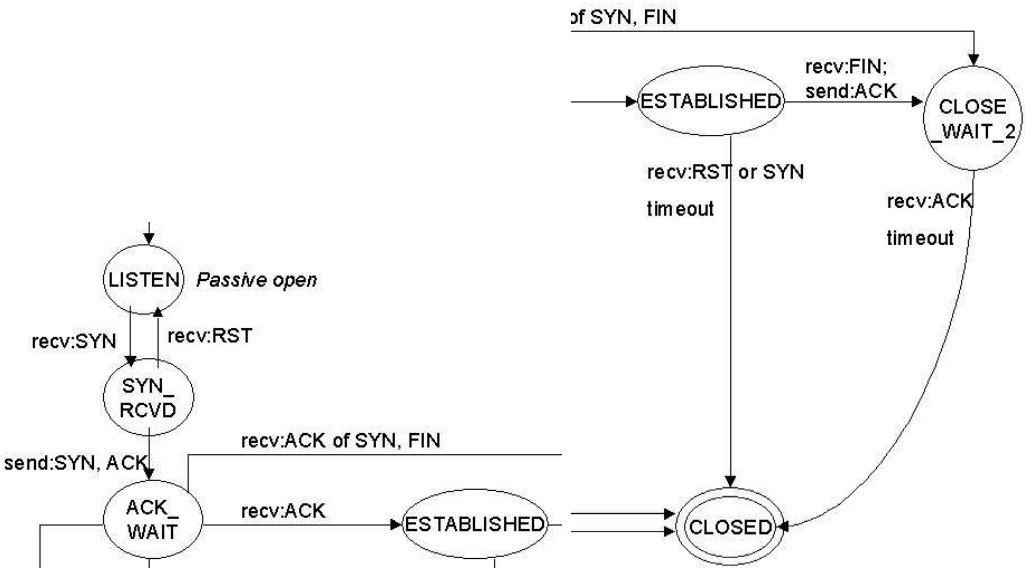

FIGURE 4.11: TCP Connection Establishment and Termination

1. **Connection Establishment**. A client sends a SYN segment specifying the port number of the server that the client wants to connect to, and the client's initial sequence number (ISN). TCP state changes from LISTEN to SYN_RCVD. The server responds with its own SYN segment containing the server's initial sequence number. The server also acknowledges the client's SYN by ACKing the client's ISN plus one. A SYN consumes one sequence number. TCP state changes from SYN_RCVD to ACK_WAIT. (see Figure 4.11). The client must acknowledge this SYN from the server by ACKing the server's ISN plus one. TCP state changes from ACK_WAIT to ESTABLISHED.

2. **Connection Termination**. When the server receives FIN, it sends back an ACK of the received sequence number plus one. TCP state changes from ESTABLISHED to CLOSE_WAIT_2 (see Figure 4.11). A FIN consumes a sequence number, just like a SYN. At this point, the server's TCP also delivers an end-of-file to the application (the discard server). The server then closes its connection,

causing its TCP to send a FIN. Then the client TCP must ACK by incrementing the received sequence number by one. TCP state changes from CLOSE_WAIT_2 to CLOSED.

4.3 SDL Modeling For Prototyping Packet Verifier

The purposes of *Packet Verifier* are validating compliance to standards, and detecting protocol anomalies. *Packet Verifier* checks the protocol header of packets, verifies packet size, checks TCP/UDP header length, verifies TCP flags and all packet parameters, does TCP protocol type verification, and analyses TCP Protocol header and TCP protocol flags. The goal of using the specification and description language (SDL) [CCITT(1992)] is not to define a formal description of the TCP verification model, but rather to provide some assurance that the TCP verification model under development are complete and perform the functions that were intended. This SDL allowed us to locate errors in requirements of *Packet Verifier*.

The specification was made by hand (Figure 4.10) first, and then using SDL is to accomplish requirement of the TCP verification model. SDL is an International Telecommunication Union (ITU) standard, based on the concept of a system of Communicating Extended Finite State Machine (CEFSM) Model [Hopcroft and Ullman(1979)]. To understand how SDL can work based on the CEFSM, it is necessary to address the dynamic semantics of the finite state machine, SDL's underlying model, and generating the TCP verification model using SDL. This rapid development of a model for testing and validating of the contained behaviour of the development verification model was useful. This process uncovered various ambiguities, unspecified transitions, and a deadlock within the draft verification model. Thus helping to ensure that at least those errors found were fixed and applied to development.

4.3.1 Dynamic Semantics Of Finite State Machines

SDL is based on the concept of CEFSMs, which communicate with each other and their common environment by signals in an asynchronous manner via possibly delaying communication paths. These signals are buffered on arrival at a process.

A finite state machine (FSM) is defined as a 4-tuple $<$ S, s_0, E, f $>$, where S is a set of states, s_0 is an initial state, E is a set of events with their parameter lists, f is a state transition relation. However, the construction of an FSM is limited by the state-explosion problem. An extended finite-state machine (EFSM) solves this problem by introducing variables in addition to explicit states of the process instance. These variables become implicit states, being able to take on a number of values themselves. Each EFSM is defined as a FSM with addition of variables to its states. EFSMs are

those defined with additional variables to states as a 5-tuple $<$ S, s_0, E, f, V $>$. Where S, s_0, E, and f as in the case of the FSM and V is a set of local variables along with their types and initial values, if any. Each state in an EFSM is defined by a set of variables, including state names. The transition T of an EFSM becomes [$<$ s, v_1, $\ldots$, v_n $>$ + $input^*$, $task^*$; $output^*$ + $<$ s', v'_1, $\ldots v'_n$ $>$], where s and s' are the names of states, $<$ v_1, v_2, $\ldots$, v_n $>$ and $<$ v'_1, v'_2, $\ldots v'_n$ $>$ are values of extended variables, n is the number of variables, "+" means coexistence, ";" means sequence of events such as tasks and outputs, and "[,]" denotes a sequenced pair. The difference between an EFSM and an FSM is that an EFSM associates each transition not only with input and output actions but also with assignment actions and conditions [Wang and Liu(1993)].

A communicating extended finite-state machines (CEFSM) includes the definitions of EFSMs and signals [Jan Ellsberger and Sarma(1997)]. There are signals, which means that channels exist. A CEFSM is defined as a 6-tuple $<$ S, s_0, E, f, V, X $>$. Where S, s_0, E, f, and V as in the case of the EFSM, and X is a set of signals. In CEFSM, signals are responsible for communicating information from within the CEFSM to other automata, some of which may be located in the environment of a system. The signals account for the observable behaviour, which is more important than the actual model for a specification. In SDL, CEFSM processes use signals to communicate with other CEFSMs and the environment.

4.3.2 SDL's Underlying Model

The language SDL is intended for the formal specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete signals. It is especially well suited for specification of communication protocols, reactive systems such as switches, routers and distributed systems. SDL has been designed for the specification and description of the behaviour of such systems, i.e., the internetworking of the system and its environments. SDL allows the hierarchical description of systems. The description starts from a construction called system, where functional blocks are inserted. A block is a component composed by one or more processes and/or other blocks. A block consists of processes connected by signal routes. A process contains a sequential behaviour and concurrency modelled by a set of processes. Each process is a CEFSM. These machines or processes run in parallel. They are independent of each other and communicate with discrete messages, called signals. A process can also send signal to and receive signals from the environment of the system. The behaviour of a state machine is characterized by a set of transitions. A transition to another state or the same state occurs whenever an input is consumed. When a process is in a state, it accepts input. This input can be a signal received by the input port or timers. When a process enters a new state, it means that a transition terminates. CEFSM enables decisions to be made in transitions based on the value associated with

a variable so that the state which follows when a specific input is consumed is not only determined by the existing state and input.

The SDL language supports two equivalent notations: the graphical notation (SDL-GR) and the textual notation (SDL-PR). The SDL-GR is a standardized graphical representation of the system. SDL elements such as system, block, process, signal etc. are drawn using standardized graphical symbols. The SDL-PR is a textual phrase representation of the SDL system, or in other words, it is a SDL source code.

4.3.2.1 Process Model

The Z.100 ITU-T standard defines that the SDL underlying model is a CEFSM (Communicating Extended Finite State Machine), where all processes are CEFSMs. For each process, a finite number of states, inputs and outputs determine its behaviour. Nondeterminism capability allows representing spontaneous transitions, which are transitions without any signal causing them. This is useful to describe unpredictable system characteristics. In SDL, only one input signal can be consumed/evaluated at each instant. This means that each input signal consumed corresponds to one state transition in an SDL description.

4.3.2.2 Communication Model

The concurrency model used in SDL allows independent and asynchronous processes operation. There is no guaranteed relative ordering of operations in distinct processes, except the ordering created by explicit synchronization among processes through the use of shared signals. Shared signal events are then the means by which a precise ordering of events in distinct process can be achieved.

The communication between processes is reliable. It is assured that the receiving process will consume every signal produced by a sender process. However, it is not guaranteed that the ordering of the signals generated by all processes is the same of their consumption. This model is adequate to describe events without precise ordering, like systems that can have their normal flow interrupted. Handshaking or unlimited queues in practice-bounded queues are used to implement the communication model. For both cases, each SDL state results in a set of protocol communication signals and area overhead to implement the protocol. This characteristic may cause large communication overhead, which can penalize the implementation.

4.3.3 Generating the Specification

The TCP verification model is specified with CEFSM and is presented in SDL in this section. A CEFSM is defined as a 6-tuple < S, s_0, E, f, V, X >, as it is mentioned above.

- S is a set of states
- s_0 is an initial state
- E is a set of events with their parameter lists
- f is a state transition relation
- V is a set of local variables along with their types and initial values, if any
- X is a set of signals

For a state, an input event, and a predicate composed of a subset of V, the state transition relation f has a next state, a set of output events and their parameters, and an action list describing how the local variables are updated.

The purpose of SDL in this project is to verify whether the simplified TCP verification model follows the standard TCP transitions. To do this, the simplified TCP verification model (Figure 4.10) was converted into a SDL specification. The CEFSM of the simplified TCP verification model is as follows:

- S = {listen, syn_rcvd, ack_wait, established, closing, close_wait_1, close_wait_2, closed}
- s_0 = listen
- E = {send(V_i, X_i), recv(V_i, X_i), timeout(V_i)}
- f : $\{f(S_i, E_i, V_i) \rightarrow (S_{i+1}, V_{i+1}, E_{i+1})\ \}$
- V = {tcp_id, tcp_seq, tcp_id_seq}
- X = { ACK, SYN, FIN, RST, SYNACK, ACKFIN}

In this SDL specification, among TCP flags, PSH and URG are not included. Timeout, and checking flags and packet sequences should be dealt with in a low-level implementation part as well.

4.3.4 SDL Creation based on the TCP Verification Model

To detect packet fragmentation, the SDL specification can recall the packet sequence and proper flag, and the low-level implementation part cooperates with this SDL specification, other flag combinations, and timeout. To build the SDL specification, Cinderella SDL [Cinderella(2003)] was used. Figure 4.12 shows the part of the StateTransition process built in SDL. Besides, all SDL-GR and -PR of the proposed TCP verification model can be found in Appendix A.

f is the state transition relation. It represents how to move from the current state to a new state given a certain action if any.

Note that ', ,' in a set of variables V means no variable changed, '{ }' in a set of events E means no specific event is required.

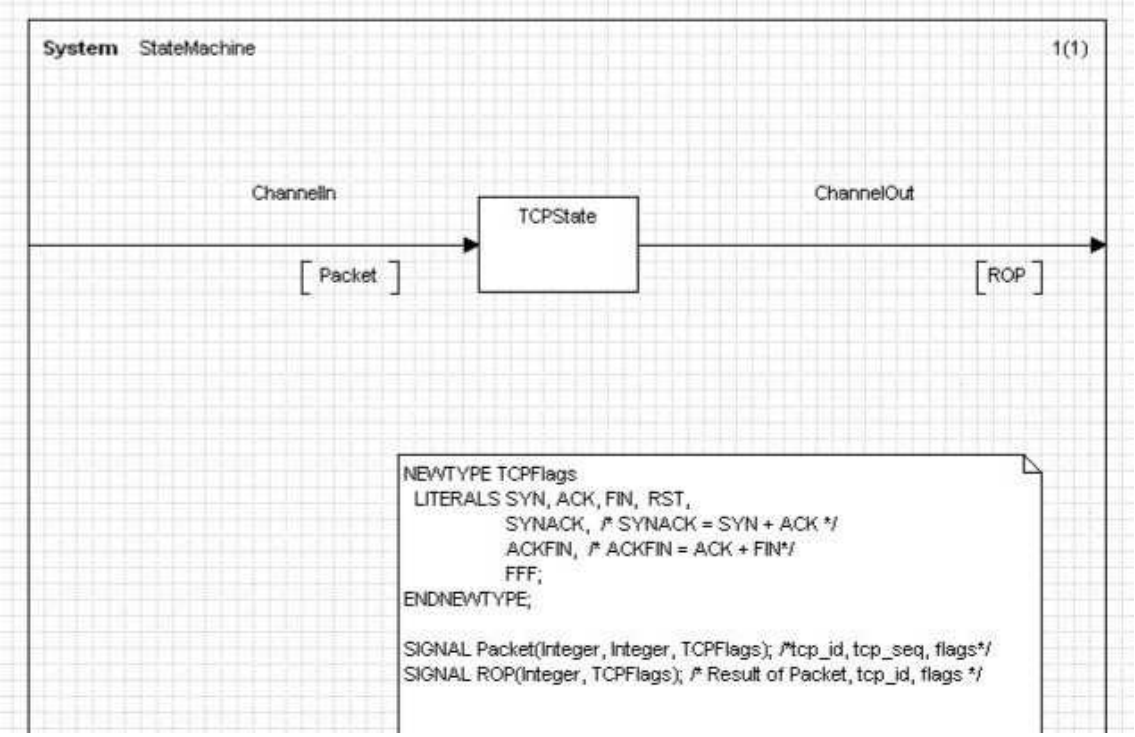

FIGURE 4.12: System of the TCP Protocol State Machine

- LISTEN State.
  f(listen, recv(tcp_id, SYN), tcp_id_seq = 0) → (syn_rcvd, tcp_id_seq = tcp_seq, send(tcp_id, SYNACK))

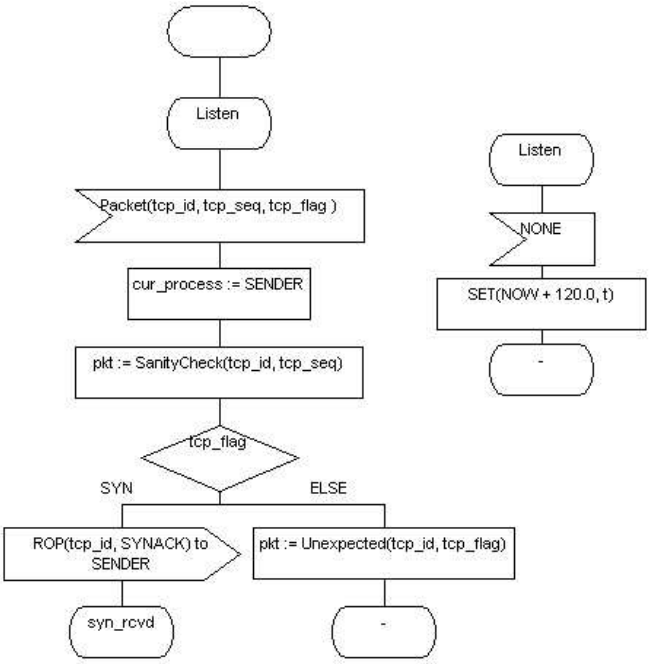

FIGURE 4.13: LISTEN State of the TCP Protocol State Machine

- SYN_RCVD State.
  f(syn_rcvd, recv(tcp_id, RST), tcp_id_seq != 0) → (listen, tcp_id_seq = 0, {}),
  f(syn_rcvd, send(tcp_id, SYNACK), tcp_id_seq != 0) → (ack_wait, , {})

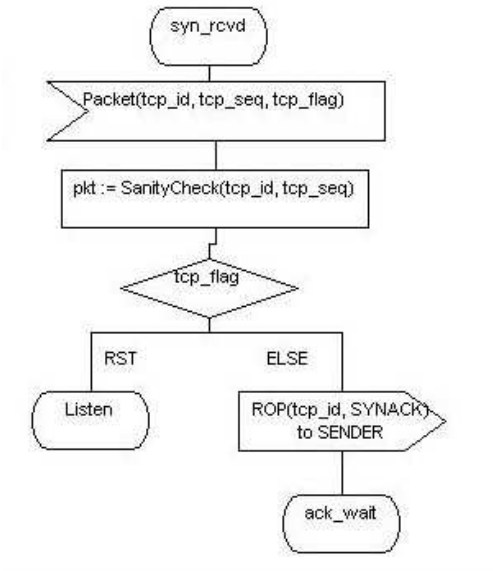

FIGURE 4.14: SYN_RCVD State of the TCP Protocol State Machine

- ACK_WAIT State.
  f(ack_wait, recv(tcp_id, ACK), tcp_id_seq != 0) → (established, tcp_id_seq = tcp_seq, {}),
  f(ack_wait, recv(tcp_id, ACKFIN), tcp_id_seq != 0) → (close_wait_2, tcp_id_seq = tcp_seq, {}),
  f(ack_wait, timeout(tcp_id), tcp_id_seq != 0) → (closed, tcp_id_seq = 0, {F}),
  f(ack_wait, recv(tcp_id, FIN), tcp_id_seq != 0) → (closing, , send(tcp_id, ACK))

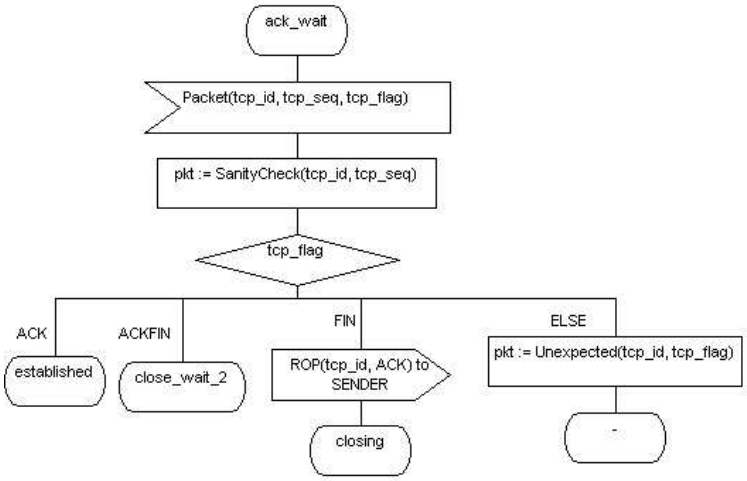

FIGURE 4.15: ACK_WAIT State of the TCP Protocol State Machine

- CLOSING State.
  f(closing, recv(tcp_id, ACK), tcp_id_seq != 0) → (close_wait_1, tcp_id_seq = tcp_id_seq, {}),
  f(closing, timeout(tcp_id), tcp_id_seq != 0) → (closed, tcp_id_seq = 0, {F})

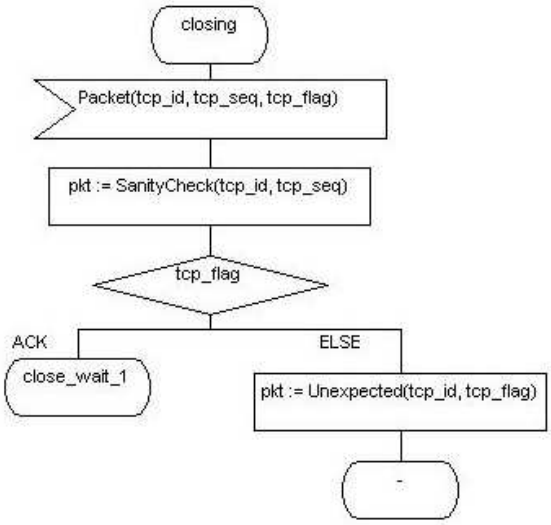

FIGURE 4.16: CLOSING State of the TCP Protocol State Machine

- CLOSE_WAIT_2 State.
  f(close_wait_2, recv(tcp_id, ACK), tcp_id_seq != 0) → (closed, tcp_id_seq = 0, {F}),
  f(close_wait_2, timeout(tcp_id), tcp_id_seq != 0) → (closed, tcp_id_seq = 0, {F})

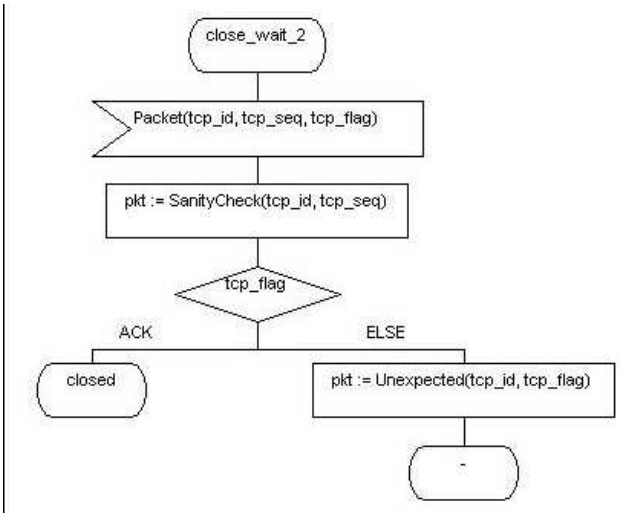

FIGURE 4.17: CLOSE_WAIT_2 State of the TCP Protocol State Machine

- ESTABLISHED State.
  f(established, recv(tcp_id, RST), tcp_id_seq != 0) → (closed, tcp_id_seq = 0, {F}),
  f(established, recv(tcp_id, SYN), tcp_id_seq != 0) → (closed, tcp_id_seq = 0, {F}),
  f(established, recv(tcp_id, FIN), tcp_id_seq != 0) → (close_wait_2, , send(tcp_id, ACK)),
  f(established, timeout(tcp_id), tcp_id_seq != 0) → (closed, tcp_id_seq = 0, {F})

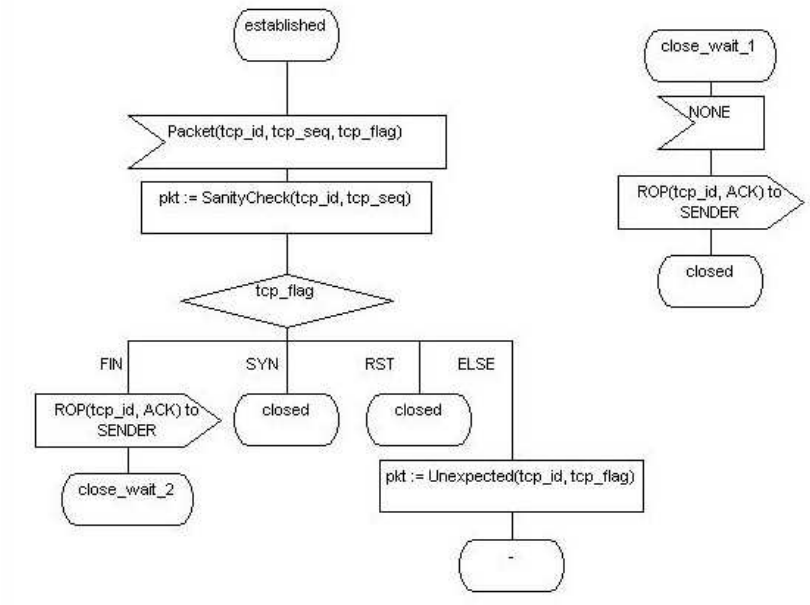

FIGURE 4.18: ESTABLISHED State and CLOSE_WAIT_1 state of the TCP Protocol State Machine

- CLOSE_WAIT_1 State.
  f(close_wait_1, send(tcp_id, ACK), tcp_id_seq != 0) → (closed, tcp_id_seq = 0, {F})

4.4 Countermeasures Against Protocol Anomaly-Based Attacks

4.4.1 Incomplete Three-way Handshake

Here countermeasures are presented using the packet verification model and Packet Verifier against incomplete three-way handshake design in the TCP mechanism.

1. **Extraneous state problem.** Consider a sequence of packets between hosts X and A. Host X sends a packet to host A, with both SYN and FIN flags set. Host A responds by sending a SYN-ACK packet back to host X. In that scenario, the TCP verification model work as follows: host A changes its state from LISTEN to SYN_RCVD because of receiving the SYN packet, and from SYN_RCVD to ACK_WAIT because of sending the SYN-ACK packet, and then because of the FIN packet, its state moves to CLOSING. Since host A receives a FIN packet, host A sends a ACK packet to host X, and then its state CLOSING wait for receiving a ACK packet from host X. Hence, if host X does not send any more packets to host A, host A waits for 10 seconds and then enters CLOSED.
2. **Simultaneous open problem.** Host X sends a SYN packet to host A and host A sends a SYN packet to host X in order to start a TCP connection. When host X receives the SYN packet from host A, it sends an ACK packet back in response. When host A receives this packet, its state of host A changed from LISTEN to SYN_RCVD and to ACK_WAIT after receiving the ACK packet from host X and replying with SYN-ACK. Hence, if host X does not send back any SYN-ACK, its state moves to CLOSED after waiting 60 seconds.

4.4.2 IP Spoofing

IP spoofing attacks are combined by incomplete three-way handshake, predictable IP identification values and then bogus IP. IP spoofing case, not by one method can solve this problem, but several combined defence measures are needed to work out.

- **Packet verification model and Packet Verifier.** IP spoofing case has to be solved by cooperation between the TCP verification model and IP packet's header analysis together. To achieve this kind of problem, we need not only to check the TCP protocol three-way handshake mechanism but also to check

packets' specific requirement in each packet whether the packet's IP address and sequence numbers are valid or forged, such as checking tcp_id as combination of source address, source port, destination address and destination port, and checking tcp_id_sequence_number, tcp_id_acknowledge_number and TTL.

- **Be un-trust relationship.** One easy way to prevent this attack is not to reply on address-based authentication. Disable all the r command such as rlogin, rsh, rcp, etc. Remove all .rhosts files and empty out the /etc/hosts.equiv file. This will force all users to use other means of remote access such as telnet, ssh, skey, etc.

- **Packet Filtering.** Make sure only hosts on the internal LAN can participate in trust-relationship. No internal host should trust a host outside the LAN. Then simply filter out all traffic from the outside the Internet that purports to come from the inside the LAN. And filter out all private network addresses (i.e., 10.0.0.0/24, 192.168.0.0/16 and 172.0.0.0/24) from the outside network.

- **Cryptographic Methods.** One of methods to deter IP spoofing is to require all network traffic to be encrypted and/or authenticated. While several solutions exist, it will be a worthwhile.

- **Initial Sequence Number Randomising.** Since the sequence numbers are not chosen randomly, this IP spoofing works. Each connection would have its own separate sequence number space. Bellovin [Bellovin(1989)] suggested the following formula:

  $$ISN = M + F(localhost, localport, remotehost, remoteport)$$

  Where M is the 4 microsecond timer and F is a cryptographic hash. F must not be computable from the outside or the attacker could still guess sequence number. Bellovin suggests F be a hash of the connection-id and a secret vector such as a random number, or a host related secret combined with the machine's boot time.

  On the other way, try to run operating systems with less predictable IP identification sequences, such as recent versions of OpenBSD, Solaris, or Linux. Solaris and Linux use peer-specific IPID sequences. In addition, Linux 2.4 zeros the IPID fields in packets with the DF (Don't fragment) bit set, since IP defragmentation is the only critical use of the ID field [Fyodor(2003)].

- **Use of IPv6.** Another way to secure communication is by using IPv6 instead of IPv4. RFC 1825 [Atkinson(1995)] specifies two new characteristics of IPv6: authentication header and encapsulation security payload. The first prevents IP address faking while the second introduces encryption for IP packet and TCP header.

4.4.3 SYN flood attack

- **Packet verification model and Packet Verifier.** Host X sends a SYN packet to host A. Host A acknowledges the SYN packet by sending a SYN-ACK packet to host X. Host X continuously sends SYN packets without responding with an ACK packet. In this case, the TCP verification model works as follows: host A changes its state from LISTEN to SYN_RCVD, and to ACK_WAIT, and then if host X does not send ACK packet but sends SYN packet continuously, its state moves to CLOSED. Since ACK_WAIT state waits for ACK packet, and SYN packet is not right one to move its state, so its state moves to CLOSED.

- **Inside operating systems.** Some operating systems stop accepting new connections if there are too many forged SYN packets at them. Many operating systems can only handle 8 packets. Linux kernels and some other systems allow various methods such as SYN cookies to prevent this from being a serious problem [Anderson(2001b)]. SYN cookies are a technique used to mitigate the effects of SYN flood attacks by choosing initial TCP sequence numbers (ISNs) that can be verified cryptographically. The server's initial sequence number is generated as follows [Bernstein(1997)]:

  - top 5 bits: t mod 32, where t is a 32-bit time counter that increases every 64 seconds.
  - next 3 bits: an encoding of an MSS selected by the server in response to the client's MSS.
  - bottom 24 bits: a server-selected secret function of the client IP address and port number, the server IP address and port number, and t.

4.4.4 Ping of Death & Land Attack

- **Packet verification model and Packet Verifier.** If each IP packet size exceeds the maximum size (64KB) or if the header length field does not exceeds the minimum size (20 bytes), the packet should be discarded. In addition, each packet's source address and port number should not have the same as the target address and port number. Otherwise, the packet should be discarded.

- **Inside operating systems.** Operating systems should check these packet fields before accepting the packet.

4.4.5 Fragment attack

- **Packet Verifier.** If a TCP fragment has non-zero offset (e.g. F0=1), then it should be discarded.

- **Router Filtering.** Router's filtering module enforces a minimum fragment offset.
- **Inside operating systems.** Operating systems can be fixed inserting some pieces of code to the kernel files to check whether the IP packet's offset is bigger than the end of the packet.

4.4.6 ICMP flood (Smurf attack) & UDP flood attack

- **Packet Filtering.** UDP packets should never be allowed to destine for system diagnostic ports from outside of administrative domain to reach intranet systems. To prevent a network from these UDP attacks, turn off broadcast addressing on all network routers that allow it unless needed for multicast features, or configure a firewall to filter the ICMP_ECHOREQUEST. In addition, UDP services could be restricted for use only within the internal network, thus keeping UDP available for network diagnostic purposes only. This prevents its unauthorized use for UDP flooding attacks. To avoid becoming the victim of the smurf attack, have an upstream firewall that can either filter ICMP_ECHOREPLYs or limit echo traffic to a small percentage of overall network traffic. Moreover, the border routers do not allow directed broadcast packets to be forwarded through their routers as a default.

Chapter 5

Email Classification For Risk Assessment

What we anticipate seldom occurs;
What we least expected generally happens.
- Benjamin Disraeli.

Email is probably the most valuable service on the Internet. The use of email for communication is constantly growing. Correspondingly, the volume of email received also grows fast. Nevertheless, it is quite vulnerable to be misused. As normally implemented [Postel(1982), Crocker(1982)], the mail server provides no authentication mechanisms. This leaves the door wide open to faked messages. RFC 822 [Crocker(1982)] does support an Encrypted header line, but this is not widely used. [1] One such misuse is by email viruses; another is by unsolicited bulk emails (UBEs) as known as spam. This project focuses on the way of propagation of UBEs, because this propagation is also a potential way of email viruses. The difference of UBE and Email virus is having malware contents or not.

UBE is defined as Internet mail or email that is sent to a group of recipients who have not requested it. A mail recipient may have at one time asked a sender for bulk email, and then later asked the sender does not send any more emails or has otherwise not indicated a desire for such additional mail; hence any bulk email sent after that request was received is also UBE.

Email viruses are defined as viruses or worms that spread using email with attachments. These email viruses include file viruses or email worms. Email worms are defined as programs that self-replicate via email. Email statistics of year 2004 about UBEs and email viruses reported by Postini [Postini(2004)] shows that the percentage of spam

[1] RFC 1040 [Linn(1988)] is for a discussion of a proposed new encryption standard for email.

emails is 71.7% (10 out of 13 messages are spam), and that 1 in 26 messages is virus-infected.

In order to determine a probability estimation which can tell us whether an email is abnormal, to find relations between email viruses and detectable knowledge, and to make a statistical relation between interesting events among incomplete data, Bayesian networks [Pearl(1988)] or probabilistic graphical models have been chosen to use. A Naive Bayesian classifier among several Bayesian network models has been used to classify packets of the SMTP protocol. Certain packet characteristics, which is analysed in [Yoo and Ultes-Nitsche(2002a)], that allow us to attach to packets probabilities of their potential maliciousness. The analysed file characteristics are used for the parameters of the Naive Bayesian classifier.

5.1 Background of Bayesian Networks

In order to deal with malicious email attachments, Bayesian networks are used as means of identifying malicious code.

5.1.1 Bayes' Theorem and Bayesian Inference

To estimate the probability of the potential maliciousness of packets, it is necessary to get some specific evidence from data. For independent events E and M, M represents a group of malicious packets, E represents specific evidence about these packets. If E and M are independent, we get:

$$\mathrm{P}(M \wedge E) = \mathrm{P}(M) * \mathrm{P}(E)$$

However, in cases where E and M are not independent. We must write:

$$\mathrm{P}(M \wedge E) = \mathrm{P}(M) * \mathrm{P}(E \mid M)$$

where, $\mathrm{P}(E \mid M)$ is the probability of the specific evidence given the malicious packets have occurred. The conditional probability of event E given event M, denoted $\mathrm{P}(E \mid M)$, is given by,

$$\mathrm{P}(E \mid M) = \frac{\mathrm{P}(E \wedge M)}{\mathrm{P}(M)}$$

Now since conjunction is commutative,

$$\mathrm{P}(M \wedge E) = \mathrm{P}(M) * \mathrm{P}(E \mid M) = \mathrm{P}(E) * \mathrm{P}(M \mid E)$$

And by rearranging we get:

$$\mathrm{P}(M \mid E) = \frac{\mathrm{P}(M) * \mathrm{P}(E \mid M)}{\mathrm{P}(E)}$$

which is known as Bayes' Theorem. We write,

- P($M \mid E$): Posterior probability, the probability of malicious packets given the specific evidence, which is what we wish to infer.
- P(E): The probability of the specific evidence; this is a measurable quantity that we get from existing data.
- P($E \mid M$): Likelihood probability, since we gain it from measurement of evidences. The probability of the specific evidence given the malicious packets. We can measure this from the case histories of the malicious packets.
- P(M): Prior probability, the probability of malicious packets that we get from existing data. Since we knew it before we made any measurements.

Prior probability should be subjective. It represents our belief about the domain we are considering, even if data has made a substantial contribution to our belief. Likelihood probability should be objective. It is a result of the data gathering from which we are going to make an inference. It makes some assessment of the accuracy of our data gathering. In practice, either or both forms can be subjective or objective. In some cases, we obtain the prior probability from statistics. For example, we can calculate the prior probability as the number of instances of a disease divided by the number of patients presented for treatment. However, in many cases this is not possible since the data is not there, and there may also be prior knowledge in other forms.

5.1.2 Naive Bayesian Classifier

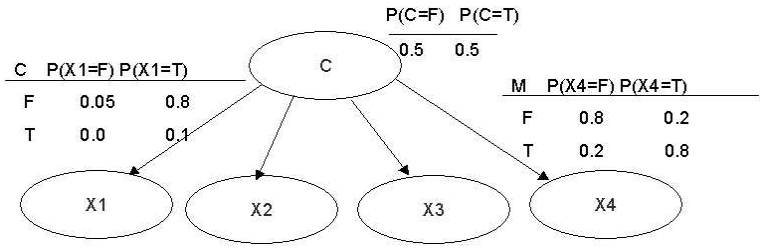

FIGURE 5.1: A Structure of a Naive Bayesian Classifier

To make inference between interesting events among incomplete data, a Naive Bayesian Classifier [Langley and Sage(1994)] has been chosen. The Naive Bayesian Classifier is a simple structure in which nodes that show the same parent node cannot have a connection between them. A Naive Bayesian Classifier is illustrated in Figure 5.1. We assume each node as a class. This Naive Bayesian Classifier represents each class with a single probability table. In particular, each table has an associated class probability $\mathrm{P}(C_i)$, which specifies the prior probability that one will observe a member of class C_i. Each table also has an associated set of conditional probabilities, specifying a probability distribution for each attribute. The Naive Bayesian classifier relies on two important assumptions [Langley and Sage(1994)]. First is a single probability table. Instances in each class can be summarized by a single probability table, and these are sufficient to distinguish the classes from one another. Another assumption is independence of attributes. The Naive Bayesian classifier requires that the probability distributions for attributes are independent of each other within each class. One can model attribute dependence within the Bayesian framework [Pearl(1988)]. However, determining such dependencies and estimating them from limited training data is much more difficult. Thus, the independence assumption has clear attractions. It is applicable in many cases.

To determine the probability of whether packets are malicious, we are getting some specific evidences from data. For independent events E and M, M represents a group of malicious packets, E represents specific evidence about these packets. When we use Bayes' theorem we have just one hypothesis and one piece of evidence. However, we have evidence from more than one source in the real world. Therefore, we apply these several evidences to Bayes' theorem, then we get:

$$\mathrm{P}(M \mid E_1 \wedge \cdots \wedge E_n) = \frac{\mathrm{P}(M) * \mathrm{P}(E_1 \wedge \cdots \wedge E_n \mid M)}{\mathrm{P}(E_1 \wedge \cdots \wedge E_n)}$$

Using this equation, the probability of malicious packets is calculated in the next section.

5.2 Generating a Naive Bayesian Classifier

5.2.1 Statistical Characteristics of Email

In order to apply real data, MRTG [Oetiker and Rand(2003)] was used for counting email messages entering the Southampton ECS department network during the past few years, and the entire virus database of Ahnlab [Ahnlab(2002)] was analysed, and then a database of email packets was built. Tables 5.1, 5.2 and 5.3 show email messages and virus & UBE numbers which were measured by MRTG. Email messages are measured by MRTG every second. To display the visual representation of this traffic, MRTG

uses the weekly representation with 30-minute average data, the monthly representation with 2-hour average data, and the yearly representation with 1-day average data. Each number of messages represents a max, an average, and a current in a year, a month, and a week. MRTG displays this number, which received data during a year, a month, and a week, in current time per day. For example, a max in a year is chosen by the maximum received number per day during one year, in a certain time when we measure. Note that the year data is based on the 1-day average, the month data is based on 2-hour average data, and the week data is based on 30-minute data. That is the reason that the current data of a year is different from that of a week and a month.

TABLE 5.1: Email messages and Virus & UBE numbers in a Year
(Date: Thursday, 25 July 2002 at 11:04)

Type	Messages	Viruses	UBEs
Max	215000	4957 (2.3%)	1881 (0.9%)
Average	8339	165 (2.0%)	706 (8.5%)
Current	9797	274 (2.8%)	1482 (15.1%)

TABLE 5.2: Email messages and Virus & UBE numbers in a Month

Type	Messages	Viruses	UBEs
Max	215000	395 (0.2%)	1881 (0.9%)
Average	8995	201 (2.2%)	1383 (15.4%)
Current	9633	223 (2.3%)	1586 (16.5%)

TABLE 5.3: Email messages and Virus & UBE numbers in a Week

Type	Messages	Viruses	UBEs
Max	109000	361 (0.3%)	1710 (1.6%)
Average	8542	293 (3.4%)	1383 (16.2%)
Current	9633	223 (2.3%)	1586 (16.5%)

According to the survey of email virus statistics, which was reported in year 2002 [Yoo and Ultes-Nitsche(2002a)], about 80% of windows file worms were transferred via email, and approx. 61% of windows file worms had .EXE file extension. In addition, an average 2% of all emails per month contained virus data (See Table 5.2), 80% of which were in an Win32 executable file format as Table 5.5 shows. In addition, Table 5.6 shows the percentage of several file extensions among windows file worm types. Most Internet viruses [2], which was detected and reported until year 2002, had Win32 executable format (about 86%) among file/network worms. In executable attachment formats, there

[2]Internet viruses are the superset of email viruses. It includes email viruses, file worms, networks worms and so on.

are .EXE (64%), .SCR (22%), .COM (6%), .PIF (22%), .BAT (3%), .VBS (2%), and Others (12%).

The email UBE statistics record [Administrators(2003)] was developed by administrators in University of Calgary. Their record was remade for the reference as presented in Table 5.4. The dates of records in this table are between 10th May 04:09:39 and 18th May 01:01:11, 2003. Besides, the blocked mails include non-existent and invalid recipients, non-existent senders' domain and DNS black lists. However, this table does not contain the DNS black list information. Note that the UBE numbers of Tables 5.1, 5.2, and 5.3 represent the number of detected UBEs by pattern matching.

TABLE 5.4: Email UBE Statistics (Total:291985)

Type	Number	Percent
normal mail	26262	8.99%
blocked mail	249622	85.4%
non-existent and invalid recipients	751	0.2%
non-existent senders' domain	248739	85.1%
invalid senders' address but not blocked	25074	8%

TABLE 5.5: Detected viruses which were entering ECS Department. (Total: 17300)
Record date: Thursday, 25 July 2002 at 11:04.

Name	Percent
W32/Sircam.A	50%
W32/Flcss	11%
W32/Klez.G	10%
W32/Navidad.B	7%
W32/Badtrans.B	5%
W32/Magistr.A	3%
W32/Navidad	2%
W32/Hybris.B	2%
Others	10%

TABLE 5.6: Classification of Windows File Worms.
Prevalence of Win32 worm via email: total 96 worms

File Format	Number	Percent
.EXE file	58	60.4 %
.PIF or .SCR file	18	18.8 %
.BAT.COM.EXE.PIF.SCR file (among these, choose 1 format)	3	3.1 %
.COM file	3	3.1 %
.VBS file	2	2.1 %
Self-executable compressed format	6	6.3 %
Other	6	6.3 %

5.2.2 Choosing Evidence Factors

A base rate may be defined as the relative frequency with which an event occurs or an attribute is present in a population [Lanning(1987), V. B. Hinsz and Robertson(1988), Ginossar and Trope(1987)]. The extent to which base rates are used appears to depend on the characteristics of the problem at hand. Base rates seem to be used more accurately when the domain or the experimental process suggests that a statistical kind of answer is warranted. If we disregard base rates, and rely on the individuality, evidences and resultant likelihood ratio of abnormal mails will be surely overestimated, especially virus emails. Therefore, considering the base rate of malicious mail, it is important to choose high accurate detectable evidences. Each evidence factor has been chosen, which is related to protocol specifications rather than data content, which can be changeable.

Emails are sent via SMTP protocol [Postel(1982)]. The usage of each protocol header was analysed to detect protocol anomalies. Email worms use spoofed email addresses under the guise of trusted sources. Email worms are sent with specific purpose not like normal mail. They are sent with one to many connections, source IP addresses are faked or spoofed, a sender's address is not valid, a sender's email address is not used by a valid domain name, a recipient's address is not used by a valid domain name, a recipient's email address is not used by IP address, or recipients are mentioned in BCc (Blind Carbon Copy) rather than To or Cc (Carbon Copy), and so on. Therefore, four factors have been selected: the header field, the sender field, the recipient field, and an attachment field of each email among the SMTP protocol. Each field of an SMTP packet to classify UBEs is normalized in a checking process in the following way:

- [Sender field] If reverse DNS domain check fails, e.g., non-existent domain, then set FALSE. If invalid senders' address, then set FALSE.
- [Recipient field] If non-existent recipient, then set FALSE. If invalid recipients'

address, then set FALSE.

Each field of an SMTP packet to classify email viruses is also normalized as follows:

- o [Header field] If there are multiple content-type headers, multiple encoding headers, or multiple non-plain headers, then set FALSE. If content-type header is MIME, then check whether or not the mail body has content-type;text/html;charset=utf-8 etc. If any match is not found, then set FALSE.
- o [Sender field] If sender address is used with an IP address rather than domain name, then set FALSE. If source address in IP packet is different from sender address, then set FALSE.
- o [Recipient field (To/Cc/Bcc)] If To does not contain a domain name (e.g. id@ip_address), then set FALSE. If To is empty and Cc includes the email address without a domain name, then set FALSE. If To and Cc are empty, then set FALSE even if Bcc contains an address with a domain name.
- o [Attachment field] If an attachment is an executable file (.EXE, .SCR, .COM, .PIF, .BAT, .VBS, and others), then set TRUE.

The raw packets of emails are only dealt and the same protocol mechanism is used for UBEs and email viruses. Therefore, UBEs' normalization can be shared with email viruses' for classification. However, to increase one's confidence, it is necessary to divide the two cases and consider them separately.

5.2.3 A Naive Bayesian Classifier Against Email Viruses

According to the surveyed prior information (Tables 5.5, 5.6, 5.1, 5.2, and 5.3), a probability to each factor was assigned. Using this data we can determine the probability of whether mail packets are malicious, denoted P(M), using the header field (H), the sender field (Fr), the recipient field (To), and the attachment field (EF) as discussed in the previous subsection. Each field of an SMTP packet is examined in a checking process to compute a probability value. This Naive Bayesian classifier (Figure.5.2) represents the probability of being malicious given several evidences such as malformed H, Fr, To, and EF. The prior probabilities of a malicious mail and the likelihood probabilities of a malicious executable attachment were assigned as discussed in Section 5.2.1.

The other prior probabilities were assigned as subjective values based on the prior probability of a malicious mail. As a result, each node has four likelihood probabilities; two (their status is set with TRUE or FALSE) different probabilities for each node given two cases (malicious mail is set with TRUE or FALSE) for the potential maliciousness

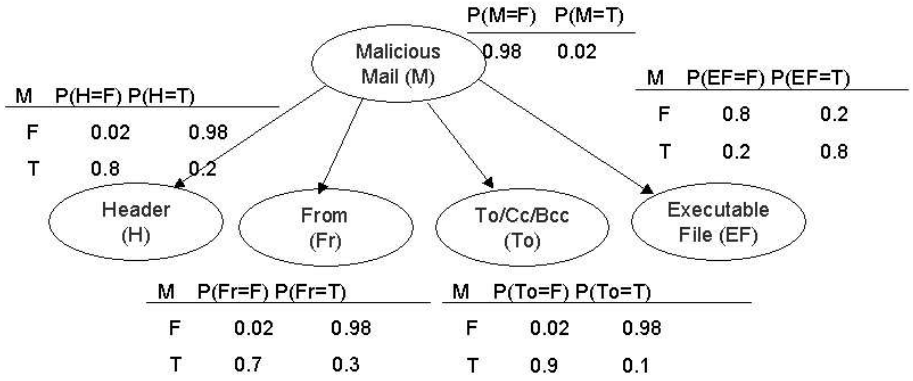

FIGURE 5.2: A Naive Bayesian classifier for detecting abnormal emails.

of a mail packet, which is denoted by M. The Naive Bayesian classifier was built as in Figure 5.2, and then an exact inference algorithm was used for Bayesian inference using MATLAB 5.3 [MATHWORKS(2003)]. The results of posterior probabilities from the Naive Bayesian classifier are in Table 5.7 .

TABLE 5.7: Results of posterior probabilities from the Naive Bayesian Network (Malicious mail M = T)

Probability of Malicious mail given evidences	Results
$P(M \mid H = F)$	0.4494
$P(M \mid H = F \wedge Fr = F)$	0.9662
$P(M \mid H = F \wedge Fr = F \wedge To = F)$	0.9992
$P(M \mid H = F \wedge Fr = F \wedge To = F \wedge EF = T)$	0.9998
$P(M \mid Fr = F)$	0.4167
$P(M \mid Fr = F \wedge To = F)$	0.9698
$P(M \mid Fr = F \wedge To = F \wedge EF = T)$	0.9923
$P(M \mid To = F)$	0.4787
$P(M \mid To = F \wedge EF = T)$	0.7860
$P(M \mid EF = T)$	0.0755

Through this result, we can predict the probability of data packets being malicious if certain evidence is given. For example, if only the header field is FALSE (meaning that it is malformed), the maliciousness probability of a current SMTP packet is 0.4494. If the sender field and the recipient field are FALSE, the maliciousness probability is 0.9698. If the sender field is FALSE, and the mail includes an executable attachment, the probability of being malicious is 0.7860. However, note that this executable attachment is not recognized as a malicious executable file yet in this moment. This Naive Bayesian Classifier just tells us, because this is an executable file, the probability of being malicious is estimated by value 0.7860. The detailed process for recognizing this executable file is the next step. Nevertheless, the value of the probability to be malicious is quite reasonable.

TABLE 5.8: Results of posterior probabilities from the Naive Bayesian classifier (Malicious mail M = T)

Probability of Malicious mail given evidences	Results
Fr=T ∧ To=T ∧ H=T ∧ EF=T	5.2013e-004
Fr=T ∧ To=T ∧ H=T ∧ EF=F	3.2524e-005
Fr=T ∧ To=T ∧ H=F ∧ EF=T	0.0926
Fr=T ∧ To=T ∧ H=F ∧ EF=F	0.0063
Fr=T ∧ To=F ∧ H=T ∧ EF=T	0.1867
Fr=T ∧ To=F ∧ H=T ∧ EF=F	0.0141
Fr=T ∧ To=F ∧ H=F ∧ EF=T	0.9783
Fr=T ∧ To=F ∧ H=F ∧ EF=F	0.7376
Fr=F ∧ To=T ∧ H=T ∧ EF=T	0.0562
Fr=F ∧ To=T ∧ H=T ∧ EF=F	0.0037
Fr=F ∧ To=T ∧ H=F ∧ EF=T	0.9210
Fr=F ∧ To=T ∧ H=F ∧ EF=F	0.4216
Fr=F ∧ To=F ∧ H=T ∧ EF=T	0.9633
Fr=F ∧ To=F ∧ H=T ∧ EF=F	0.6212
Fr=F ∧ To=F ∧ H=F ∧ EF=T	0.9998
Fr=F ∧ To=F ∧ H=F ∧ EF=F	0.9969

TABLE 5.9: the truth table of malicious email

Fr	To	H	EF	f
T	T	T	T	0
T	T	T	F	0
T	T	F	T	0
T	T	F	F	0
T	F	T	T	0
T	F	T	F	0
T	F	F	T	1
T	F	F	F	0
F	T	T	T	0
F	T	T	F	0
F	T	F	T	1
F	T	F	F	0
F	F	T	T	1
F	F	T	F	0
F	F	F	T	1
F	F	F	F	1

The results of posterior probabilities from the Naive Bayesian classifier are in Table 5.8. In addition, to build an ordered Binary decision diagram from the email classifier in Figure 5.2, the posterior probabilities were extended from Table 5.7 to Table 5.8. The result of applying a threshold 0.9 is in Table 5.9. These results is used to classify email viruses whether they confirm maliciousness, depending on whether the probability of

maliciousness given the evidences is no less than a certain threshold, say, 0.9. Using this threshold, an email decision diagram is built in the next section.

5.3 Generating OBDDs For Email Classifier Model

5.3.1 Ordered Binary Decision Diagrams

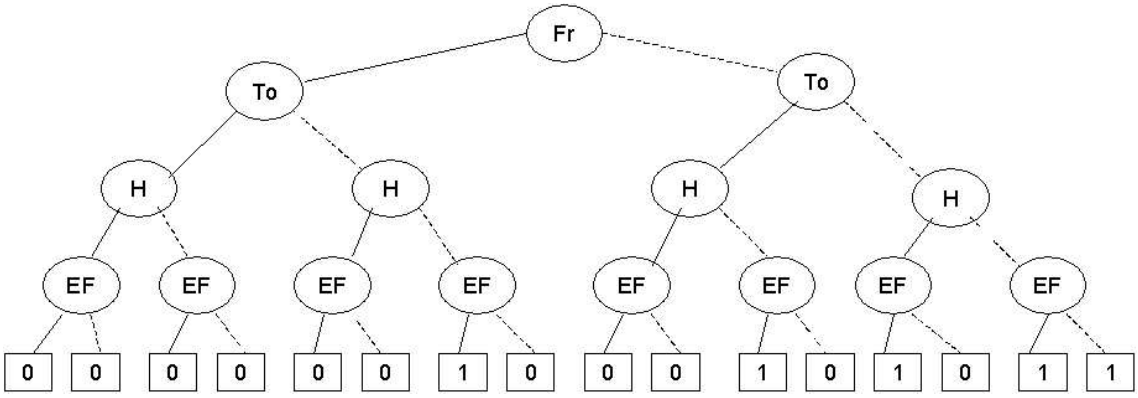

FIGURE 5.3: Decision Tree Representation of Malicious Email

Binary decision diagrams (BDDs) [Bryant(1986)] have been recognized as abstract representations of Boolean functions. A BDD represents a Boolean function as a rooted, directed acyclic graph. As Figure 5.3 illustrates, a representation of the function $f(Fr, To, H, EF)$ defined by the truth table Table 5.9, leads to the special case where the graph is actually a tree. Terminal nodes of out-degree zero are labelled 0 or 1, and a set of variable nodes v of out-degree two are used. The two outgoing edges are given by two functions $low(v)$ corresponding to the case where the variable is assigned 0, and $high(v)$ corresponding to the case where the variable is assigned 1, these are shown as dotted and solid lines, respectively. A variable $var(v)$ is associated with each variable node.

The key idea of OBDDs [Bryant(1992)] is that by restricting the representation, Boolean manipulation becomes much simpler computationally. A BDD is OBDD if on all paths through the graph the variables respect a given linear order $x_1 < x_2 < ... < x_n$, such as $Fr < To < H < EF$. An OBDD is reduced if no two distinct nodes u and v have the same variable name and low- and high-successor, i.e., $var(u) = var(v), low(u) = low(v), high(u) = high(v)$ implies $u = v$, and no variable node u has identical low- and high-successor, i.e., $low(u) \neq high(u)$ [Bryant(1992)].

5.3.2 OBDD Representation From The Naive Bayesian Classifier

Reduced OBDDs [Bryant(1992)] provide compact representations of Boolean expressions. They are all based on the crucial fact that for any function $f : B^n \rightarrow B$, there is

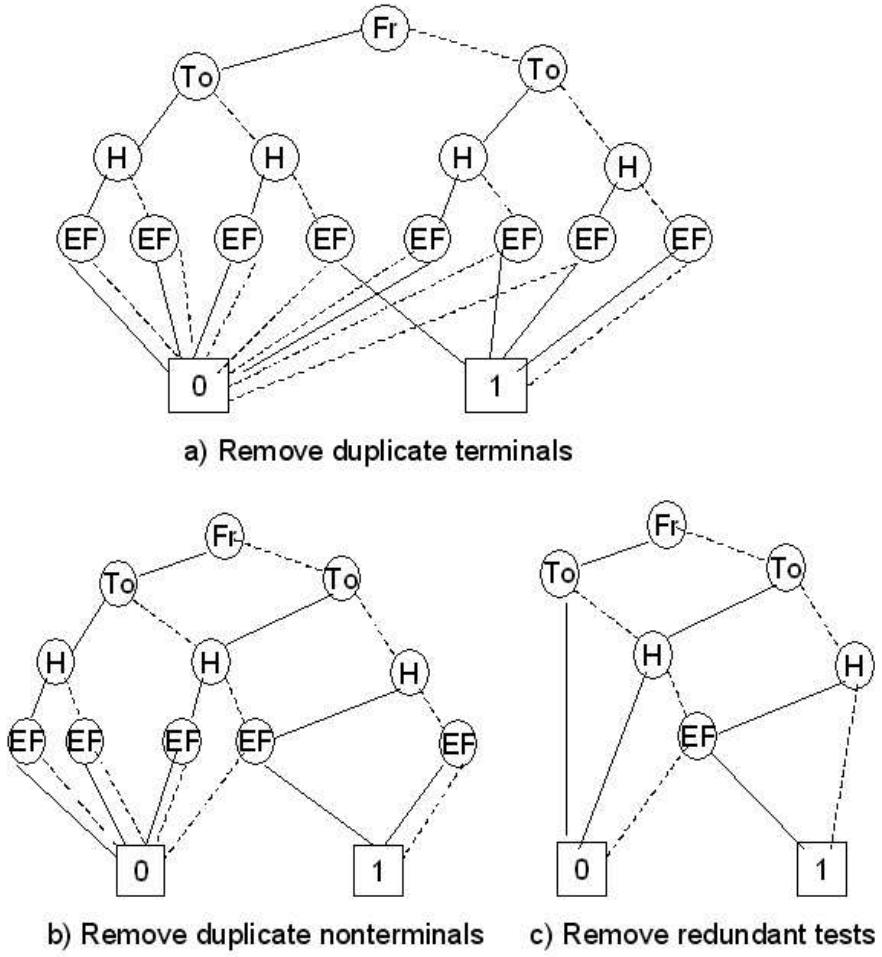

FIGURE 5.4: The removal process to build a reduced OBDD.

exactly one reduced OBDD representing it, for a given ordering. This means, in particular, that there is exactly one reduced OBDD for the constant true and constant false function on B^n: the terminal nodes 1 and 0. Hence, it is possible to test in constant time whether a reduced OBDD is constantly true or false. Furthermore, OBBDs are good to reason about the properties of any Naive Bayesian classifier. Specifically, when any Naive Bayesian classifier is represented by an OBDD that is tractable in size even given an intractable number of instances. The size of the graph representing a function can depend heavily on the ordering of the input variables.

Table 5.9 represents the truth table of malicious email using a threshold 0.9. Figure 5.3 represents the classifier induced by the Bayesian network using Table 5.9. To build an OBDD from this decision tree, transformation rules [Bryant(1992)] were applied, e.g., remove duplicate terminals, remove duplicate nonterminals, then remove redundant tests (see Figure 5.4).

The transformation rules are defined in [Bryant(1992)] as follows:

- **Remove Duplicate Terminals:** Eliminate all but one terminal vertex with a

given label and redirect all arcs into the eliminated vertices to the remaining one.

- **Remove Duplicatate Nonterminals:** If nonterminal vertices u and v have $var(u) = var(v), low(u) = low(v), high(u) = high(v)$ then eliminate one of the two vertices and redirect all incoming arcs to the other vertex.
- **Remove Redundant Tests:** If nonterminal vertex v has $low(v) = high(v)$, then eliminate v and redirect all incoming arcs to $low(v)$.

After this reduction of the decision tree, an reduced OBDD was produced as in Figure 5.5. This OBDD represents the naive Bayesian classifier induced by the network in Figure 5.2 with probability threshold 0.9, with respect to variable order (Fr, To, H, EF).

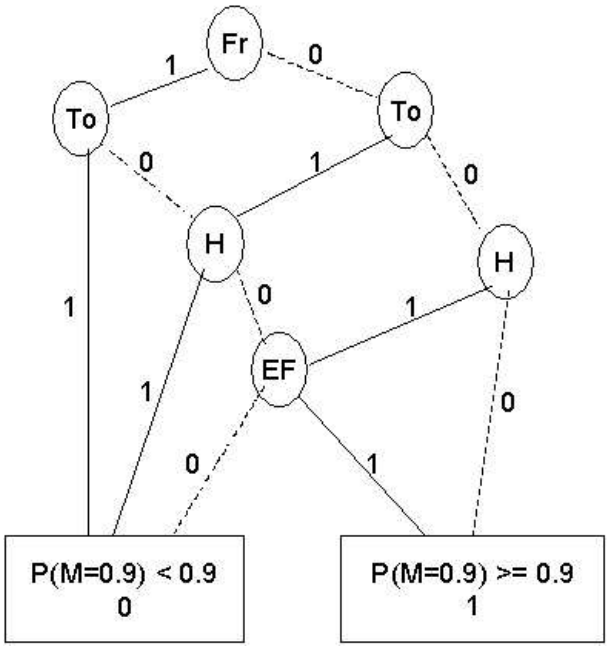

FIGURE 5.5: A reduced OBDD of Email Viruses

5.3.3 Email Classifier With OBDDs

A truth assignment to a Boolean function B is the same as fixing a set of variables in the domain of B, i.e., if X is a Boolean variable in the domain of B, then X can be assigned either 0 or 1 (denoted [X $\rightarrow$ 0] and [X $\rightarrow$ 1], respectively). Let X $\rightarrow Y_1, Y_2$ denote the if-then-else operator. Then X $\rightarrow Y_1, Y_2$ is true if either X and Y_1 are true or X is false and Y_2 is true; the variable X is said to be the test expression. More formally, we have:

$$X \rightarrow Y_1, Y_2 = (X \wedge Y_1) \vee (\neg X \wedge Y_2)$$

All operators can easily be expressed using only the if-then-else operator and the constants 0 and 1. Hence the operator gives rise to a new kind of normal form.

- **Definition** An If-then-else Normal Form (INF) is a Boolean function built entirely from the if-then-else operator and the constants 0 and 1 such that all tests are performed only on variables.

This is known as the Shannon expansion of t with respect to $u + v$. From the Shannon expansion we get that any Boolean function can be expressed in an If-then-else normal form (INF) by iteratively using the above substitution scheme on t. The ordering of the variables, corresponding to the order in which the Shannon expansion is performed, is encoded in the BDD [Bryant(1986)]. If abnormal mail classifier is denoted by t, UBEs part is by u and Email virus part is by v, we by $t[0/u+v]$ denote the Boolean expression obtained by replacing $u + v$ with 0 in t and then it is not hard to see that the following equivalence holds:

$$t = u + v \rightarrow t[1/u + v], t[0/u + v].$$

Then the abnormal mail classifier t is true if either u or v are true, which means that this classifier can say that a mail is abnormal by considering the UBEs part or email virus part in the detected raw packets. The truth table of the abnormal mail classifier is in Table 5.10.

TABLE 5.10: the truth table of UBE and that of abnormal mail

Fr	To	f	UBE	Virus	f
F	F	1	F	F	0
F	T	1	F	T	1
T	F	1	T	F	1
T	T	0	T	T	1

Like building the OBDD for email viruses, an OBDD for UBEs also can be built. However, considering the previous survey and examination of UBEs in Table 5.4, about 85% of email was blocked and 85.3% of emails was reported as non-existent or an invalid reason. Apart of this statistics, non-existent or invalid senders/recipients are also protocol anomalies. Therefore, two factors have been chosen, i.e., a sender denoted by Fr and a recipient by To. A simple Boolean function can be created using the NAND Boolean operator as follows. The truth table of this Boolean function is in Table 5.10. The OBDD representations of the UBEs classification and abnormal mail classification is in Figure 5.6.:

$$u = \neg(Fr \wedge To)$$

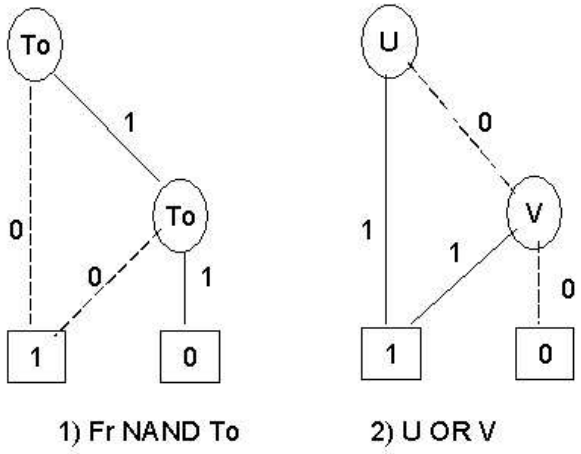

FIGURE 5.6: OBDD representation of UBEs and abnormal mail classification

The UBEs classifier u is true if either a sender field Fr or a recipient field To are false as in Figure 5.6, which means that this classifier can say this mail is an UBE by either a sender field or a recipient field are malformed or wrong.

For the email virus part v, a Boolean function can be built according to Figure 5.5 in the following way:

$$v = (Fr \wedge \neg To \wedge \neg H \wedge EF) \vee (\neg Fr \wedge To \wedge \neg H \wedge EF) \vee (\neg Fr \wedge \neg To \wedge H \wedge EF) \vee (\neg Fr \wedge \neg To \wedge \neg H)$$

The email virus classifier v is true if four factors Fr, To, H, EF are joining towards terminal (1) as in Figure 5.5, which means that this classifier can estimate that this mail contains an email virus using these facts; although a sender field is correct, a recipient field and a header field are wrong and there is an attachment in the mail, or although a recipient field is correct, a sender field and a header field are wrong and there is an attachment in the mail, or a sender field and a recipient field are wrong even though a header field is ok and there is an attachment, or a sender field, a recipient field and a header field are all wrong whether there is an attachment or not.

The results on email classification which is presented in this chapter will be used in the intelligent firewall implementation as presented in Section 7.1.

Chapter 6

Virus Visualization & Recognition

That all knowledge begins with experience,
there is indeed no doubt ....
but although our knowledge originates with experiences,
it does not all arise out of experience.
- Immanuel Kant.

Virus software is probably the most widely discussed class of computer threat. To qualify as a virus a program must meet one special criteria [Chantico(1992)]: the code in the program must be able to replicate or copy itself so as to spread through the infected machine or across to other machines.

In the general case, malicious software detection is theoretically infeasible. In the specific case of searching for a particular malicious code instance, it is not only possible, but performed daily by anti-virus software. Thus, we have good commercial solutions to detecting known malicious code instances. However, the problem of determining whether software has malicious functionality is not decidable in the general case [Rubin and Geer(1998)]. That is, we cannot look at a given application and, in general, decide whether it contains code that will result in malicious behaviour. This is equivalent to the halting problem in computer science theory, which states that there is no general-purpose algorithm that can determine the behaviour of an arbitrary program [Davis and Weyuker(1983)]. Aside from the halting problem, the property of being malicious depends to a large extent on the beholder and the context. For example, a disk-formatting program might be exactly what the user wants and therefore is not considered malicious, though when embedded in a screensaver unbeknownst to the user, it can be considered malicious. Thus, we cannot develop an algorithm to decide maliciousness.

The major spread of email viruses in 2005 is through using file worms sent via emails. In September 2005, the top ten viruses (53.2%) of detected viruses were file worms spread by either email attachments and/or network shares [Sophos(2005)]. Nonetheless, the original method of virus infection must not be ignored. These "classical" viruses are the scope of this chapter. The classic virus-detection techniques look for the presence of a virus-specific sequence of instructions, called a virus signature, inside the program: if the signature is found, it is highly probable that the program is infected. For example, the Win95.CIH/Chernobyl virus is detected by checking for the following hexadecimal sequence [Wang(1998)]: "E800 0000 005B 8D4B 4251 5050 0F01 4C24 FE5B 83C3 1CFA 8B2B".

Apart from commercial anti-virus solutions to detecting known viruses in virus-infected files, what options are we left with in addressing unknown viruses? The research presented in this thesis differs from traditional approaches to the malicious code problem in that it does not attempt to define or identify malicious behaviour. Instead, the research focuses on structural characteristics of malicious executable code. This approach allows for methods of examining any application, whether previously known or unknown, in order to determine if it has been tampered with since its original development. Such tampering usually takes the form of an embedded virus or Trojan horse that is activated during subsequent executions of the program.

To detect virus patterns including unknown ones in Windows virus-infected executable files, this project has chosen the unsupervised learning, especially the Self-Organizing Map (SOM) [Kohonen(1995)]. This chapter explains and presents the non-signature based virus detection approach using SOM. Virus-infected files cannot hide the presence of the virus through the SOM projection. As no knowledge (no signature etc) about a virus is required to detect it, the SOM-based approach can detect not only known viruses but also unknown ones. As defence in depth strategy, we can build much more secure systems by accompanying traditional virus-detection techniques based on virus signatures with the non-signature based virus-detection technique for unknown viruses.

6.1 Background of Self-Organizing Map

This section does not intend to give a complete theoretical foundation of Self-Organizing Maps (SOMs) [Kohonen(1995)]. In order to understand better of SOMs, some background of SOMs and relevant part of SOM theories are addressed. (More details can be found in books on SOMs such as [Kohonen(1982), Kohonen(1988), Kohonen(1995), Hinton and Sejnowski(1999), Haykin(1999)].)

SOM [Kohonen(1995)] is an unsupervised neural network, which does not require that the user specifies desired outputs, in contrast to the supervised neural network, which require that one or more outputs are specified in conjunction with one or more inputs

to find patterns or relations between data [Haykin(1999)]. SOM is also a feed forward neural network which uses an unsupervised training algorithm, and through a process called self-organization, configures the output units into a topological representation of the original data [Kohonen(1982)].

The SOM algorithm is based on competitive learning. SOM reduces multi-dimensional data to a lower dimensional map or grid of neurons [Hinton and Sejnowski(1999)]. It provides a topology preserving mapping from the high dimensional space to map units [Kohonen(1988)]. Map units or neurons usually form a two-dimensional grid and thus the mapping is a mapping from a high dimensional space onto a simple topology, e.g. rectangular or hexagonal. The property of topology preserving means that a SOM groups similar input data vectors on neurons: points that are near each other in the input space are mapped to nearby map units in the SOM. The SOM can thus serve as a clustering tool as well as a tool for visualizing high-dimensional data.

SOM consists of two layers of processing units [Kohonen(1995)]: the first is an input layer containing processing units for each element in the input vector; the second is an output layer or grid of processing units that is fully connected with those at the input layer. The number of processing units at the output layer is determined by the user based on the initial shape and size of the map that is desired. Unlike other neural networks there is no hidden layer or hidden processing units [Haykin(1999)].

6.1.1 SOM Algorithm

The principal goal of the SOM algorithm developed by Kohonen [Kohonen(1982)] is to transform an incoming signal pattern of arbitrary dimension into a one- or two-dimensional discrete map, and to perform this transformation adaptively in a topological ordered fashion. When an input pattern is presented to a SOM network, the winning output unit will be the unit whose incoming connection weights are the closest to the input pattern in terms of Euclidean distance [Kohonen(1995)]. Thus, the input is presented and each output unit competes to match the input pattern. The output that is closest to the input pattern is declared the winner. Often starting from randomised weight values, the output units slowly align themselves such that when an input pattern is presented, a neighbourhood of units responds to the input pattern. The connection weights of the winning unit are then adjusted, i.e. moved in the direction of the input pattern by a factor determined by the learning rate.

As training progress, the size of the neighbourhood around the winning unit and the learning rate will decrease [Kohonen(1995)]. Initially large numbers of output units will be updated, but as the training proceeds, smaller and smaller numbers are updated until at the end of the training only the winning unit is adjusted. SOM creates a topological mapping by adjusting not only the winner's weights, but also adjusting the weights of

the adjacent output units in close proximity to the neighbourhood of the winner. So, not only is the winner adjusted, but also the whole neighbourhood of output units is moved closer to the input pattern.

There are three basic steps involved in the application of the algorithm after initialisation, namely, sampling, similarity matching, and updating. These three steps are repeated until the map formation is completed. The algorithm is summarized as follows based on Kohonen's book [Kohonen(1988)]:

1. **Initialisation**. Choose random values for the initial weight vectors $w_j(0)$. The only restriction here is that the $w_j(0)$ be different for $j = 1, 2, ..., N,$, where N is the number of neurons in the lattice. It may be desirable to keep the magnitude of the weights small.

2. **Sampling**. Draw a sample x from the input distribution with a certain probability; the vector x represents the sensory signal.

3. **Similarity Matching**. Find the best-matching (winning) neuron $i(x)$ at time n, using the minimum-distance Euclidean criterion: $i(x) = argmin_j||x_n - w_j||, j = 1, 2, ..., N$

4. **Updating**. Adjust the synaptic weight vectors of all neurons, using the update formula

$$m_j(n+1) = \begin{cases} w_j(n) + \eta(n)[x(n) - w_j(n)], & j \in \Lambda_{i(x)}(n) \\ w_j(n), & otherwise \end{cases}$$

   where $\eta(n)$ is the learning-rate parameter, and $\Lambda_{i(x)}(n)$ is the neighbourhood function centred around the winning neuron $i(x)$; both $\eta(n)$ and $\Lambda_{i(x)}(n)$ are varied dynamically during learning for best results.

5. **Continuation**. Continue with step 2 until no noticeable changes in the feature map are observed.

The learning process involved in the computation of a feature map is stochastic in nature, which means that the accuracy of the map depends on the number of iterations of the SOM algorithm. Moreover, the success of map formation is critically dependent on how the main parameters of the algorithm, namely, the learning-rate parameter η and the neighbourhood function Λ_i are selected. Unfortunately, there is no theoretical basis for the selection of these parameters.

6.1.2 SOM's Properties

The SOM has properties of both vector quantization and vector projection algorithms.

6.1.2.1 Quantization

The quantization from the N training samples to M prototypes reduces the original data set to a smaller, but still representative, set to work with. Further analysis is performed primarily, or at least initially using the prototype vectors instead of all of the data. Using the reduced data set is only valid if it really is representative of the original data. When the number of prototypes approaches infinity and neighbourhood width is very large, numerical experiments have shown that the results are relatively accurate even for a small number of prototypes [Kohonen(1999)]. While the connection between the density of prototypes of SOM and the input data has not been derived in the general case, it can be assumed that SOM roughly follows the density of the training data. The primary benefit of using a reduced data set is that the computational complexity of subsequent steps is reduced. Another benefit of vector quantization is that it usually involves averaging of data samples, thus removing zero-mean noise and reducing the effect of outliers.

6.1.2.2 Projection

Since the prototype vectors of SOM have well-defined positions on the low-dimensional map grid, SOM is a kind of vector projection algorithm. The projection of a data sample can be defined to be the index b or location r_b of its BMU on the map grid. The projection is discrete as it can only get as many values as there are map units. Therefore, different vectors may be projected to the same point. Also, since the shape of the map is defined beforehand, information of the global shape of the data manifold is lost. The topological ordering of map units depends primarily on the local neighbourhood, which is defined on the map grid. Since there are more map units where data density is high, the neighbourhood in these areas becomes smaller as measured in the input space. Thus, the projection tunes to local data density.

6.2 File Format & Virus Types

The research differs from traditional approaches to the malicious code problem in that it does not attempt to define or identify malicious behaviour. Instead, the research focuses on structural characteristics of malicious executable code. This approach allows for methods of examining any application, whether previously known or unknown, in order to determine if it has been tampered with since its original development. The initial target file format considered is Microsoft Windows executable format as about 80% of detected viruses or worms were Windows executable files, in particular, approximately 61% had an ".EXE" file extension (reference to section 5.2.1).

6.2.1 Parasitic Viruses

Parasitic viruses are all viruses, which change the content of target files while transferring copies of them. The files themselves remain completely or partly usable [Pfleeger(1997)]. The most common method of virus incorporation into a file is by appending the virus to the end of the file as shown in Figure 6.1. In this process, the virus changes the header of file in such way that the virus code is executed first. In Windows and OS/2 executables (NewEXE - NE, PE, LE, LX), the fields in the NewEXE header are changed. The structure of this header is much more complicated than that of a conventional DOS EXE file, so there are more fields to be changed: the starting address, the number of sections in the file, properties of the sections etc. In addition to that, before infection, the size of the file may increase to a multiple of one paragraph (16 bytes) in DOS or to a section in Windows and OS/2. The size of the section depends on the properties of the EXE file header [Pfleeger(1997)].

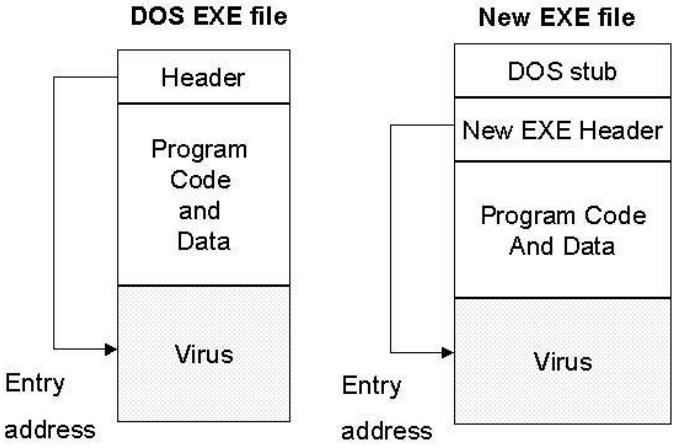

FIGURE 6.1: Virus positions in EXE and document files.

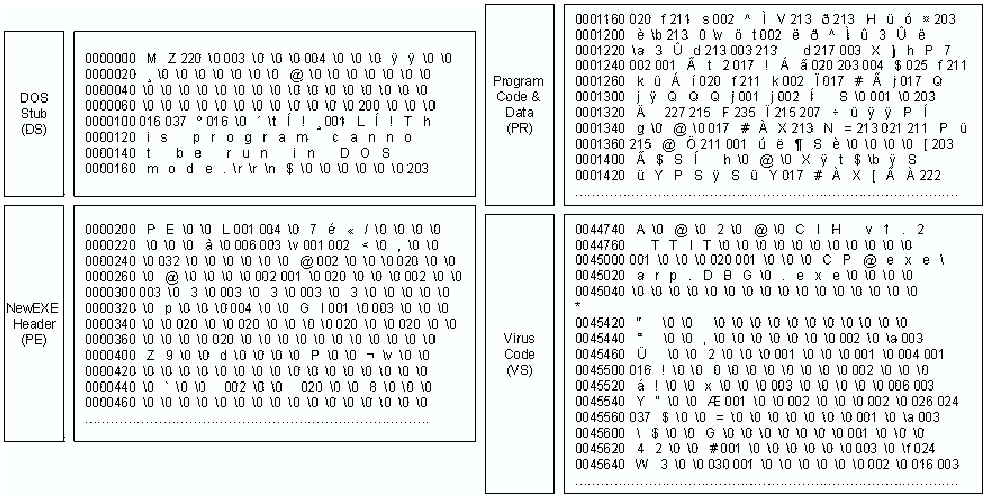

FIGURE 6.2: Structure of Windows Executable file

The structure of the data of a virus-infected file is similar to what can be seen in Figure

6.2, where the positions are octal number. When several virus-infected files have been examined, most files have the same size of the DOS stub (say, 128 bytes), and varying other parts. Apart from the virus code, only the PE (portable executable format) header is filled with quite similar character patterns, containing text, data, source and relocation information. The program code and data contain compiler-generated character sequences. Looking at Figure 6.2, the character sequence of the virus code differs from the other program code, which means that the program code and the inserted virus code have different data characteristics, e.g. because the original code was compiled as one solid piece of code, and the virus is injected only afterwards.

TABLE 6.1: Location starting point information of Test files (Unit: bytes)
Note. DOS stub always starts from 0000.

Virus file	PE header	Program Code	Virus Code
Win95.CIH Ver 1.2	128	576	11632
Win95.CIH Ver 1.3	128	624	14256
Win95.CIH Ver 1.4	128	576	1968
Win95.Boza.A	128	1024	2016
Win95.Boza.C	256	1536	3072
Win32.Apparition	128	1024	38912
Win32.HLLP.Semisoft	128	1024	41360

Table 6.1 shows the test data file information based on the file structure. If these four different areas in the virus-infected file are identified, they can be used to prove the virus visualization correct as done in the next section. Each different area will be labelled, e.g. DS for the DOS stub, PE for the NewEXE header, PR for the program code & data, and VS for the virus code. These labels are only used to identify where each part of the file will be located in the SOM projection, in order to show that the virus code is located in an area of close neighbourhood; the labels are not necessary for SOM training and visualization.

6.2.2 Macro Viruses

Another virus type appearing in Windows systems is macro virus. Macro viruses are programs written in macro languages of programs such as Microsoft Word and Excel as presented in Figure 6.3. To propagate, such viruses use the capabilities of macro languages and with their help transfer themselves from one infected file, e.g. document or spreadsheet, to another. Microsoft Word Version 6 and 7 allows to encrypt macros in documents [Kaspersky(2000)]. Therefore, some word viruses are present inside the infected documents in an encrypted, execute-only form.

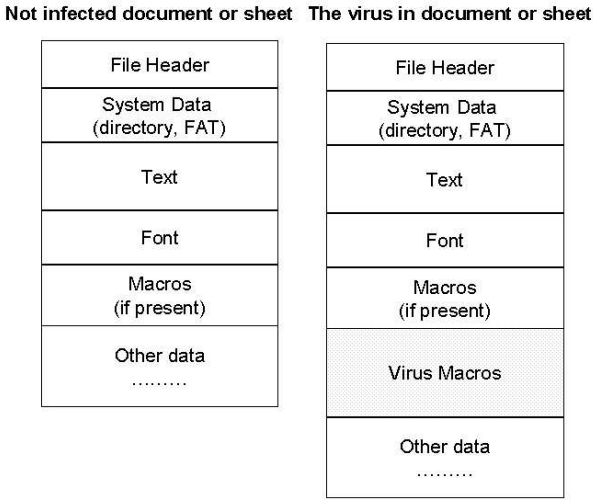

FIGURE 6.3: Macro virus position in an infected document.

6.2.3 Polymorphic viruses

Polymorphic viruses cannot or can with significant effort be detected using virus signatures. Polymorphic viruses try to remain undetected by changing their structure with each infection. There is no unique signature, which anti-virus programs can search for. Some polymorphic viruses even use different encryption techniques with every infection. In this research, polymorphic viruses have not been dealt with separately, because these polymorphic viruses can be included in parasitic and macro viruses, and they can be handled in the approach similarly to parasitic and macro viruses. The assumption of polymorphic viruses is that they are somehow inserted in the executable file, and this inserted virus code, as described in section 6.2.1, will differ again from the original program code. Thus, whether encrypted or polymorphic, the virus code can be distinguished from original program code as it is injected into an intact, kind of homogeneous program file.

6.3 SOM Training & Visualization

The SOM Toolbox 2.0 [Esa Alhoniemi and Vesanto(2002)], a software library for MATLAB 6.0 [MATHWORKS(2003)] was used under Linux to visualize virus-infected files. The visualization experiments were carried out under Linux as a precaution against infection with any of the viruses used (they were all Windows viruses).

6.3.1 Data Preparation for SOM Training

To train a SOM, a virus-infected file's binary data was converted into a table of numerical values. In general SOM data, each row of the table is one data sample, which means

the entire table consists of n different data samples. The columns of the table are the variables. The items in one row are values of these variables from the data set. The number of variables depends on features of data and was chosen to be 8 in the experiments.

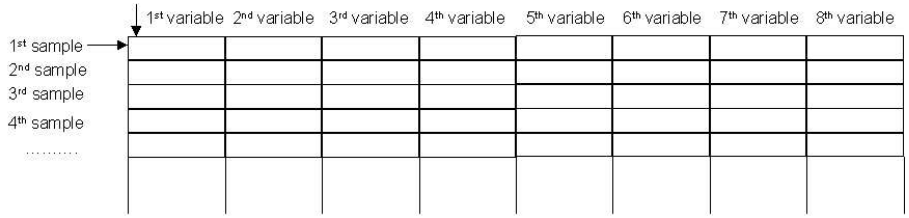

FIGURE 6.4: Table-format data: fixed length and the sample variables

The 8 variables are presented in Figure 6.4. For the virus-detecting SOM, multiple bytes were given to each variable to reflect data characteristic features. Each row of the table is one row of one (possibly virus-infected) file's binary data, and the table is a transformed form of the (possibly virus-infected) file (I call it this transformed form of the file "file under test").

To match the table structure with multiple bytes per variable in the table, a common octal-dump open source program (command name "od" in Linux) was rewritten to remove the front offset information from the dump output, transforming a binary file into a short-integer typed data format. The short integer format was chosen in order to keep the range of numerical values relatively small (in C, short integers range from -32768 to +32767). In this transformation, 4 bytes are assigned to each of the 8 variables per row, i.e. each input sample of the SOM will contain 32 bytes of the file under test. To summarize, the table consists of single 8×4-byte data samples representing n (number of rows) different portions of the file under test without overlapping data. It should be noted that this is an unusual way of using a SOM (it is not trained with n different data samples but it is "trained" with n fractions of the same sample).

6.3.2 Visualization Method

There are many different methods of displaying SOMs. I use the unified distance matrix or Umatrix since the shape of the Umatrix corresponds to the density structure of the input data, and the location of the best-matching prototypes corresponds to the topography of the input data. Therefore, the SOM "trained" with a file under test reflects the file's structure in the Umatrix.

The Umatrix represents the map as a regular grid of neurons, which can be visualized easily. Every neuron gets a numeric value assigned that corresponds to its local density in the input-space: the average distance between its prototype and the neighbouring nodes' prototypes. A low value corresponds to a high local density, a high value to a

low local density. For visualization purposes, these values can be converted easily into a colour scheme: a light (low value) colour corresponds to a high local density, and a dark (high value) colour to a low local density, or vice versa.

6.3.3 Process of SOM training and visualization

Figure 6.5 illustrates the process of SOM training and visualization. Each step in Figure 6.5 is described subsequently.

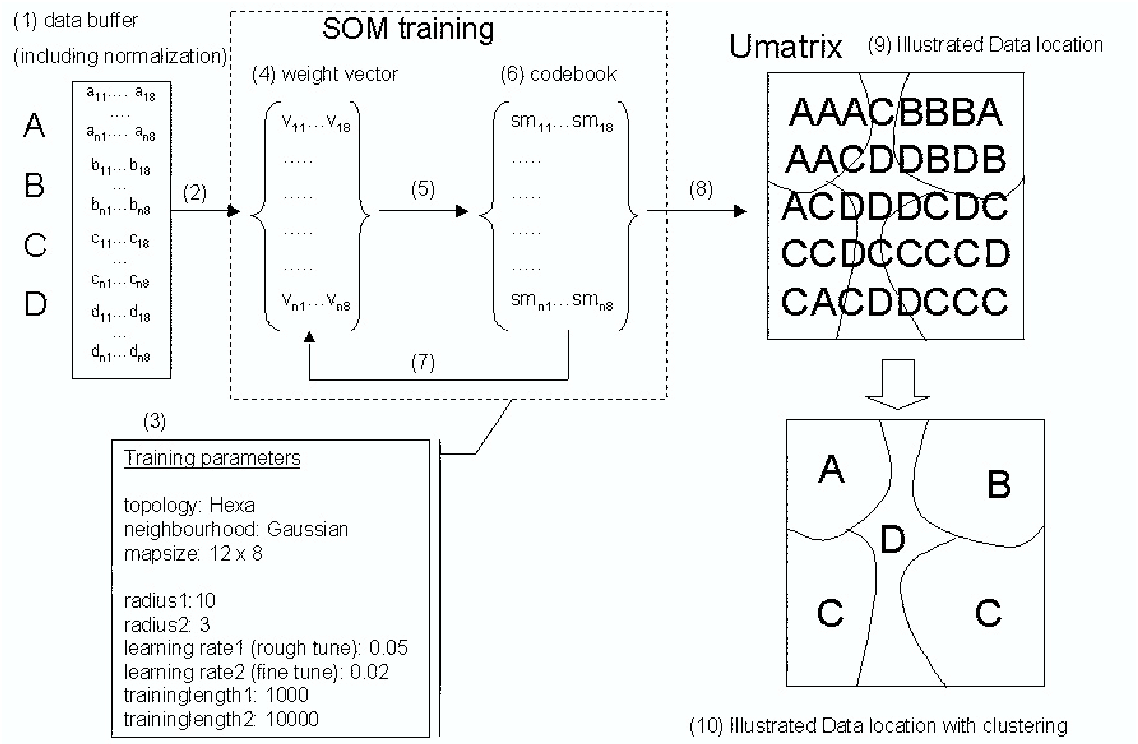

FIGURE 6.5: Process of SOM training and visualization

1. Data buffer represents the input data in a table format. The size of the input data is arbitrary. In this example we assume we can partition it into four parts labelled A, B, C and D, each representing n rows of the input data (the entire input data consists of $4n$ rows, where n is an arbitrary number). A consists of n rows containing the data designated by $a_{11}, a_{12}, ..., a_{18}$, $a_{21}, a_{22}, ..., a_{28}$, ..., and $a_{n1}, a_{n2}, ..., a_{n8}$. Similarly, B, C, and D are defined. There is no data overlap between any of these parts. Finally, in this step, the entire data is normalized and then used to train the SOM.

2. At the beginning of the SOM training, the entire data is quantized and in an initial training step, eigenvectors to each entity are calculated. There exists a random and a linear initialisation phase in the training of SOMs. In this approach, the

linear initialisation is used, which uses a linear mapping whose eigenvectors are used in the initialisation phase.

3. Using certain parameters in the SOM training phase, namely hexagonal topology, Gaussian neighbourhood, and particular map size values, each row value of the weight vector is calculated and updated until finding best matches to the input data.

4. The weight vector is used for updating the vector value in each point (SOM cell).

5. Once the entire weight vector structure is calculated, the values are saved in the so-called codebook vector. In each step, the winning entry is found in the codebook using Euclidean distance by going through the list of all weight vectors, each time computing the distance between the codebook and the input entry from the weight vector.

6. Once the fine tune is performed, the codebook is fixed and consists of best-matched vector values.

7. The weight vector and the codebook are referenced and updated for rough tune and fine tune.

8. The Umatrix is one of methods to visualize a SOM. The Umatrix visualizes the codebook vector values.

9. The Umatrix visualization reflects the quantized data. Similar data is located in a close neighbourhood to one another, producing an area of similar data density.

10. Having given different sections of the data with different name allows representing the location of each group of the data in the Umatrix. The purpose of the illustration is to cluster around the data in the Umatrix about which we know where it was located initially in the input data, aiming at identifying what the SOM does to that data. This cluster of data represents data's close neighbourhood. Like this, virus data will turn out to be represented in close neighbourhood to one another, which can be distinguished from the other data.

6.3.3.1 SOM projection

Figure 6.6 shows a simple example to illustrate the process of SOM training and visualization. It shows that if there is similar "isolated" data in a file, it will be distinguished from remaining data during the training of the SOM.

1. There is a very small input data set in the data buffer. For simplicity, identical data rows are labelled with one of the letters A, B, C, D and E. A, B, C, and D represent exactly one row of data, E represents the last 4 rows of identical data.

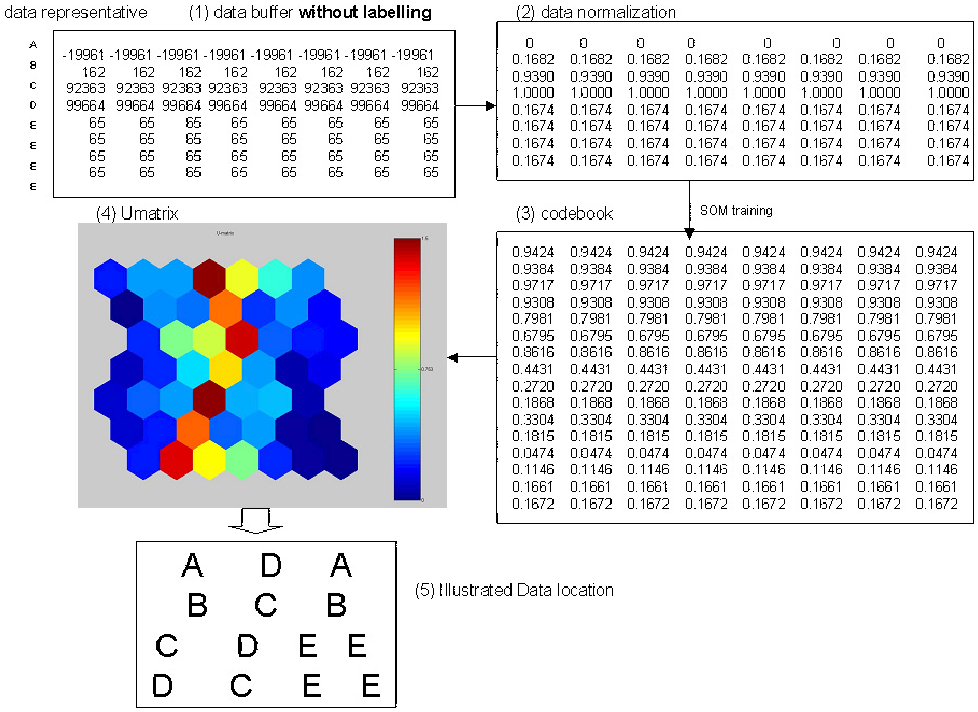

FIGURE 6.6: Example of SOM training and visualization

We will use this example to present how the Umatrix reflects the existence the area of identical data labelled with E.

2. This is the input data after normalization. The entire data structure is not changed - E still represents rows of identical data.

3. After the SOM training process, the codebook is produced as presented in Figure 6.6 (3).

4. The Umatrix presents the codebook vector values. In the example, only the bottom-right neurons are in close neighbourhood to one another when compared to other neurons.

5. The illustration presents how SOM training located the data based on their density. The "E data" appears in an area of neurons with a close neighbourhood relation to one another, compared to other data, because relatively many identical data instances (labelled with E) were fed into the SOM to train it. That caused the SOM to calculate a high density of data belonging to E, meaning that the neurons are in close neighbourhood to one another. The illustrated data location using the labels was created after identifying each neuron in the Umatrix with its original label. We will call the way the original data is distributed in the UMatrix the "SOM distribution" (explained in the next section using Figure 6.7) with data labels.

It is important to note that labels are used only to identify where a previously identified part of the input data will be distributed. They are not a part of the training process itself. By using such a labelling, an area in the UMatrix will be shown, so-called the *virus mask*, represents indeed virus data. In the virus detection system, which is developed in Janus project, data is obviously unlabelled, as any prior knowledge is not given about where, if at all, virus data and other fraction of the data are located.

6.3.3.2 SOM Distribution

This technique was used to identify which data part was located where. Using this SOM distribution, the virus part was identified and proved that the virus detection approach using SOM was reasonable. Note that this SOM distribution was only used for the purpose of proving where input data distributed.

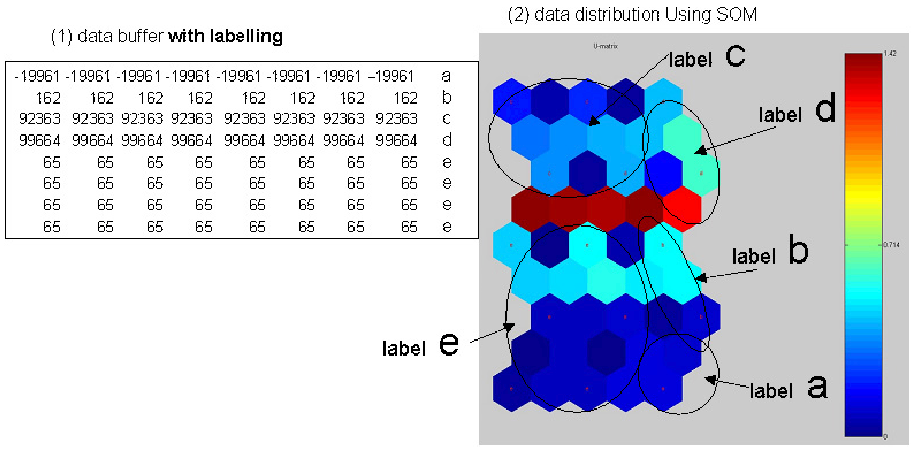

FIGURE 6.7: Example of SOM distribution with labelled data

The way to produce this distribution of data as presented in Figure 6.7 is a bit different from normal SOM training. That is the reason why the output of the Umatrix (Figure 6.7 (2)) is different from Figure 6.6 (4).

1. The same data buffer, which was used in Figure 6.6, holds labelled data at the end of each row. In this case, each row belongs to that label, e.g. the first row belongs to label 'a'.

2. This SOM distribution represents each label's data density. Each label's data is located together. This method is used for identifying each data's location and density. The SOM distribution proves that the E part is located together in Figure 6.6 (4) and illustrated in like Figure 6.6 (5).

6.4 Virus Visualization Using MATLAB

This section discuss how viruses can be visualized using the SOM-based approach. In the SOM projection, a particular pattern will be identified, which signals the presence of a virus. This pattern will be called the *virus mask*. Table 6.2 presents the test file information.

TABLE 6.2: Test file information

variants of the virus	file size
Before infection by CIH1.2	11,632 Bytes
Before infection by CIH1.3	14,256 Bytes
Before infection by CIH1.4	1,968 Bytes
Win95.CIH Ver 1.2	19,536 Bytes
Win95.CIH Ver 1.3	36,864 Bytes
Win95.CIH Ver 1.4	4,608 Bytes
Before infection by Win95.Boza.A	2,016 Bytes
Before infection by Win95.Boza.C	3,072 Bytes
Win95.Boza.A	12,408 Bytes
Win95.Boza.C	16,384 Bytes
Before infection by Win32.Apparition	38,912 Bytes
Win32.Apparition	96,239 Bytes
Before infection by Win32.HLLP.Semisoft	41,360 Bytes
Win32.HLLP.Semisoft	59,904 Bytes
Macro.Word97 mw97a	60,928 Bytes
Macro.Word97 mw97b	53,760 Bytes
Macro.Word97 mw97c	68,608 Bytes
Macro.Word97 mw97d	54,272 Bytes
Macro.Word97 mw97e	62,976 Bytes
Macro.Word97 mw97f	65,536 Bytes
Macro.Word97 mw97g	76,288 Bytes

6.4.1 Initialisation

To train a SOM, a virus-infected file's binary data was converted into a table of numerical values as it is explained in Section 6.3.1. The file under test of WIN95CIH12.exe was fed into a SOM and the SOM was initialised like below.

d = som_read_data('Win95CIH12.exe.dat');

6.4.2 Normalisation

Although binary data was converted to decimal short format, the range of data is still big. We need to represent each data item using a value between 0 and 1. To do this, SOM normalization function was used to normalize data to values between 0 and 1. The 'range' option implies values are normalized between [0,1].

sd = som_normalize(d, 'range');

6.4.3 Creation

To create, initialise and train a SOM, there are some values we need: map size, topology, neighbourhood, radius values, learning rates, and training length. For example, mapsize: 12x8, topology : hexagonal, neighbourhood: Gaussian, radius values: for rough tune 10, for fine tune 3, learning rate values: for rough tune 0.05, for fine tune 0.02, trainlength values: for rough tune 1000, for fine tune 10000.

sm = som_make(sd, 'init', 'lininit', 'msize', [12,8], 'algorithm', 'batch', 'lattice', 'hexa', 'neigh', 'gaussian', 'training', 'long');

6.4.4 Visualization

To visualise the trained SOM, there is a 'som_show' function. Unified distance matrix or Umatrix, is a method of displaying SOMs. Umatrix represents the map as a regular grid of neurons. The size and topology of the map can readily be observed from the picture where each element represents a neuron. Every node of the SOM's grid gets a numeric value that corresponds to its local density in the input-space: the average of distances between its prototype and the neighbouring nodes' prototypes. A low value corresponds to a high local density. A high value corresponds to a low local density. The numeric values are shown as shades of grey.

The location of the best-matching prototypes corresponds to the topography of the input data. The shape of the Umatrix corresponds to the density structure of the input data. This can be done in order to discover the spatial structure of a given set of data samples. First, when generating an Umatrix, a distance matrix between the reference vectors of adjacent neurons of two-dimensional map is formed. Then, some representation for the matrix is selected, e.g., a grey-level image. The colours in the figure have been selected so that the lighter the colour or lower of the colour value between two neurons, the smaller the relative distance between them.

som_show(sm, 'umat', 'all');

In this function, 'umat' implies show Umatrix with all variables, e.g., the whole variables in the codebook.

6.5 Result of Virus Visualization

6.5.1 Example Case: Win95 CIH Virus

The Win95.CIH/Chernobyl [Samamura(1998)] is a Windows95/98/NT specific parasitic virus infecting Windows PE (Portable Executable) files, and has about 1 Kbytes of length. As this virus was one of the most famous viruses, which appeared periodically from 1998 to 2004 in slightly different variants, it is chosen to discuss here. Figure 6.8 shows the SOM projection of two Windows executable files before Win95.CIH infection. The SOM projections of the tested Windows executables are different to one other, because they are different executable files. However, after Win95.CIH infection, the SOM projections of the files develop a similar pattern as presented in Figure 6.9: a dark area (navy coloured in the coloured map, representing an area of SOM cells where each cell has a short distance to its neighbouring cells). The easily identifiable dark spot is so-called the *virus mask*. It is the pattern that signals the presence of a virus in the file under test.

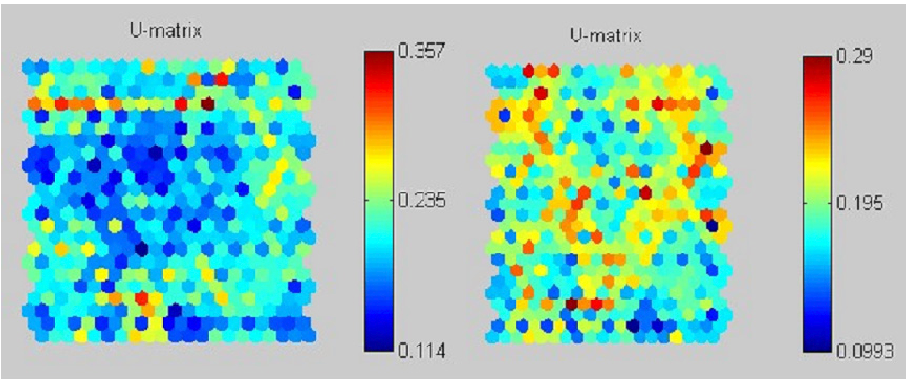

FIGURE 6.8: SOM projections of two different Windows EXE files before infection

Figure 6.9 shows two SOM projections after training the SOM with two Win95.CIH (versions 1.2 and 1.3) infected Windows executable files respectively. Each Win95.CIH/Chernobyl *virus mask* has an obvious location: top of the centre. Although the tested Windows executable files were different, the SOM projections of CIH virus-infected files look similar and have the same sort of projection map.

To prove that the *virus mask* represents indeed the CIH virus code, the SOM was trained in another experiment with data to which labels DS (DOS stub), PE (NewEXE header), PR (program code), and VS (virus code) were attached. The labels reflect the structure of an infected file as was presented in Figure 6.2. In order to attach the

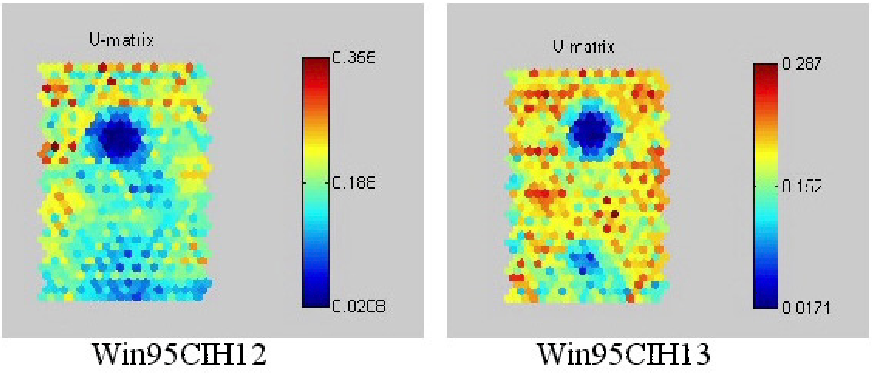

FIGURE 6.9: SOM projections of Windows EXE files infected by Win95.CIH viruses

labels, the structure of the infected file was examined "by hand". When the data set was produced, to each row a label was added as shown in Figure 6.7 (1). The result of the SOM distribution with labelled data is presented in Figure 6.10. This SOM distribution has grouped the same labels together. Therefore the figures of the SOM distribution with labels are quite different from the normal SOM projection (Figure 6.9).

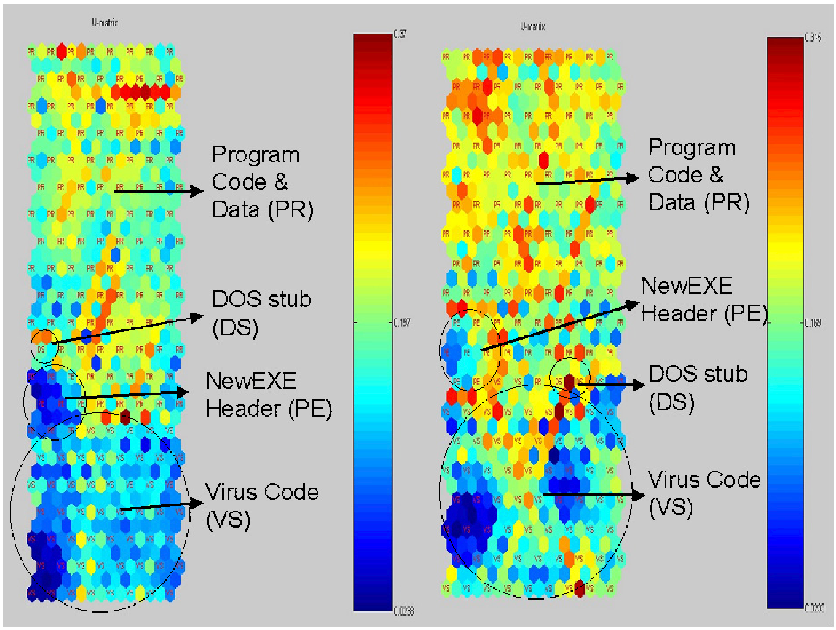

FIGURE 6.10: Virus SOM Distribution of CIH 1.2 and 1.3 viruses.

The circles in Figure 6.10 identify the different data areas except for the program code area (PR). PR is simply represented by the remaining area. As Figure 6.10 shows, there are two parts where cells have a smaller distance to their neighbouring cells: PE (NewEXE header) and VS (Virus Code). Even though PE shows an area of smaller distances between two SOM cells, VS dominates the area of small cell distances (black area in grey-scale, navy-coloured area in colour print), proving that the *virus mask* identified in Figure 6.9 represents indeed virus code.

6.5.2 Example Case: Win95 Boza Virus

Boza virus [Service(1996)] is the first known virus infecting Windows Portable Executable (PE) files, such files are used by Windows 95/NT. However, Boza does not infect machines running the Microsoft Windows NT operating system. It searches for EXE files, checks the files for PE signature, and then creates in the EXE file a new section named ".vlad", and writes its code into that section.

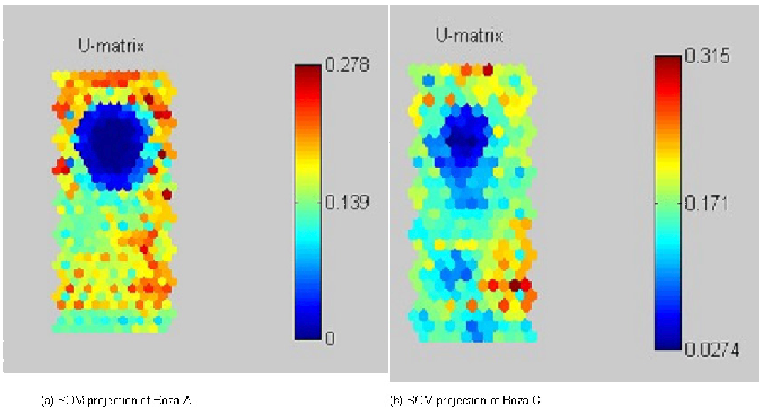

FIGURE 6.11: SOMs of Win95 Boza.A and Boza.C viruses.

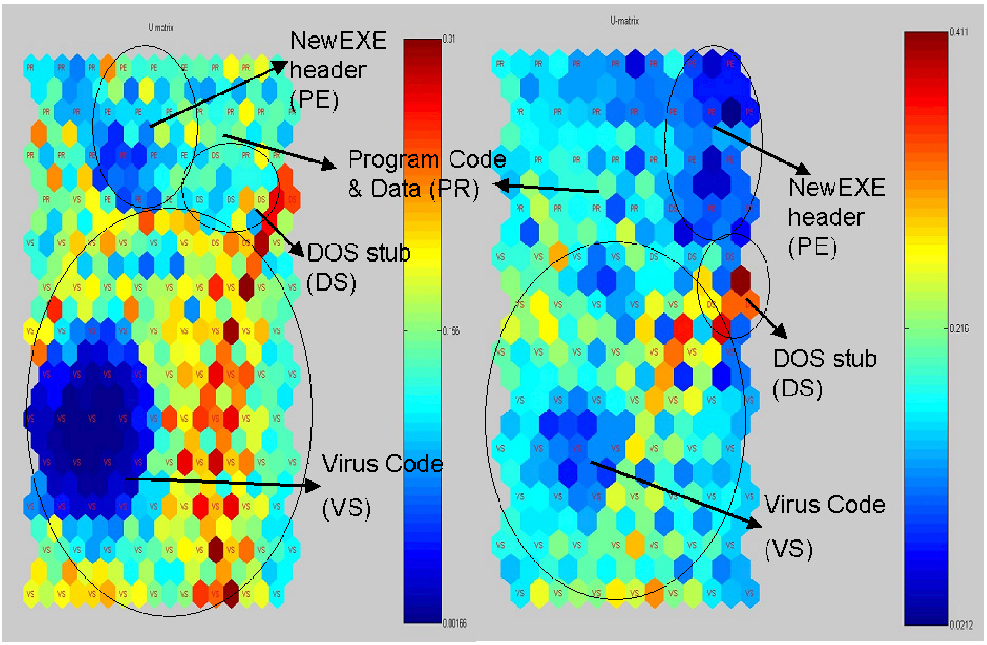

FIGURE 6.12: SOM Distribution of Boza.A and Boza.C viruses.

Two different variants of Boza virus were selected and trained. Figure 6.11 shows the Boza virus projection: the majority of lower valued colour in the upper centre represented the Boza virus code. To prove this, SOM distribution was made with labels as shown in Figure 6.12. As it was expected, the majority of smaller distance area was the Boza virus code. Although the NewEXE header code also had small distances, it did not represent the majority.

6.5.3 Example Case: Win32.Apparition

Win32.apparition virus [KASPERSKY(2003)] has a very unusual structure. The main part (about 60K) is the virus code (virus routines and C runtime library), text strings, icon and other data used by the virus while installing and spreading. The next block (3.5K) contains a packed (with LZ method) MS Word template - a Word macro virus. The third block (21K) contains packed (by LZ) virus source code. And the last block (3K) contains a resources file that is used when the virus runs the Borland C compiler.

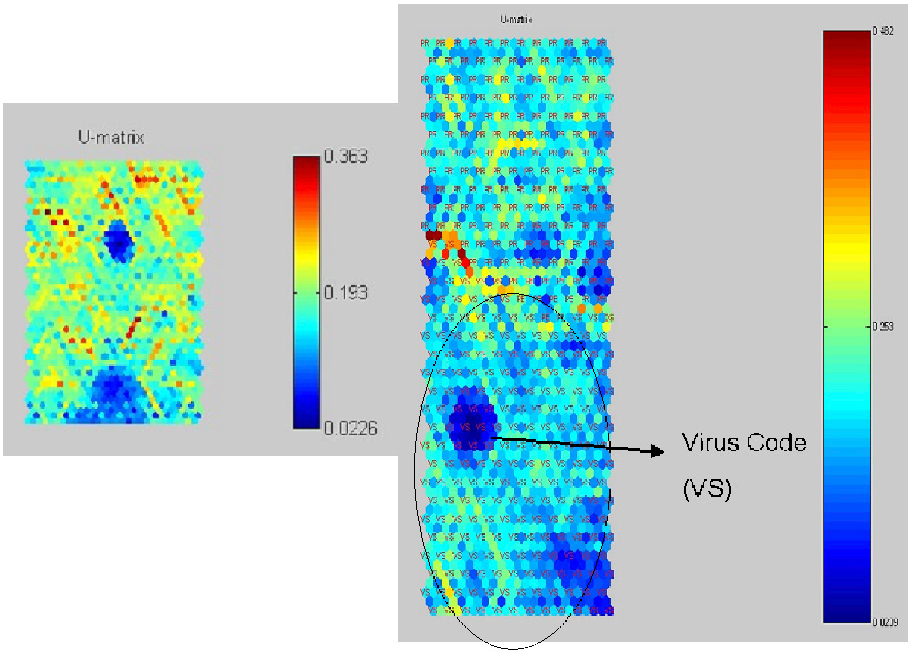

FIGURE 6.13: SOM projection and distribution of WinNT apparition virus.

Like other virus SOM projections, this virus SOM also had the *virus mask*. Figure 6.13 shows the projection of Win32.apparition virus and the projection with labels for the distribution respectively. In addition, this virus code had an unusual structure: the distribution of the virus code was quite similar to the program code and the data parts. Nevertheless, the majority of the smaller distance area represented the virus code.

6.5.4 Example Case: Win32.HLLP.Semisoft

Win32.HLLP.Semisoft virus [McAfee(2001)] is an unusual file infector which infects files under Windows 95/NT. Figure 6.14 shows the SOM projection of Win32.HLLP.Semisoft virus and the SOM distribution with labels for the data distribution respectively. Like the previous virus SOMs, this SOM also has the *virus mask* and the majority of the smaller distance area represented the virus code.

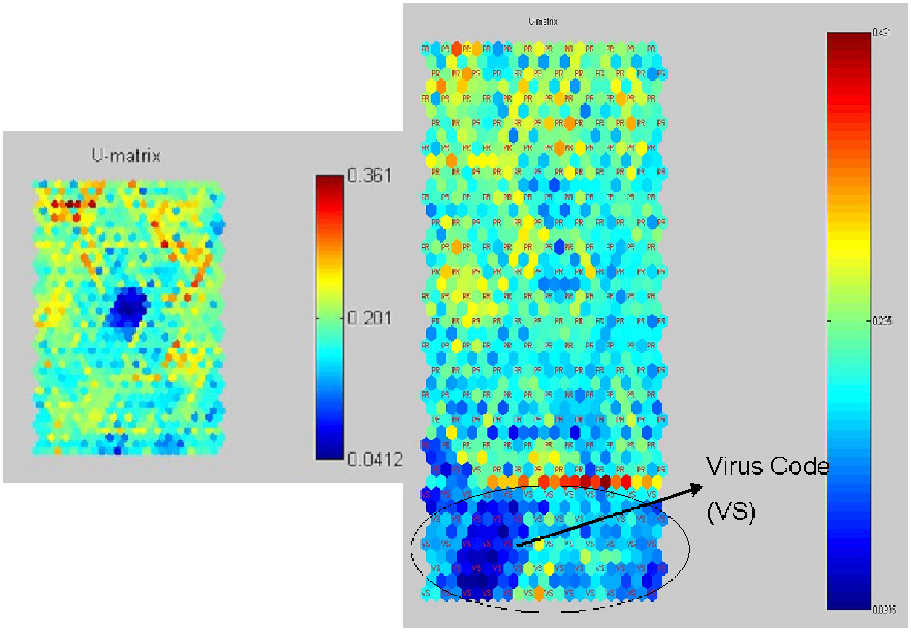

FIGURE 6.14: SOM projection and distribution of Win32.HLLP.Semisoft virus.

6.5.5 Example Case: MacroWord97.Mbug Virus

When SOM was trained with macro viruses, the result was as in Figure.6.15. The Macro.Word97.Mbug virus is a class macro virus for Word97 documents and templates. The infected file is an MS Word document file. Variants of the Macro.Word97.Mbug virus are also presented in Figure.6.15 after training by the SOM.

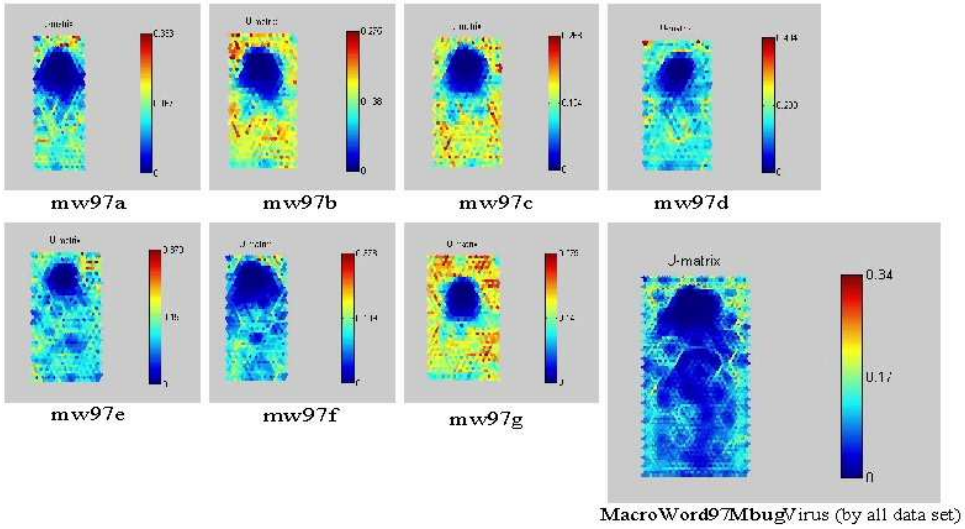

FIGURE 6.15: SOMs of Word97 file infected by Macro viruses.

Although the figures of the SOM projections look very similar to the *virus mask* in the CIH example, the result of analysing the SOM data distribution (Figure 6.16) showed that the majority of the top-centre located data was not virus code (VS) part but were labelled OD and SD (these reflect the particular structure of Word documents and will

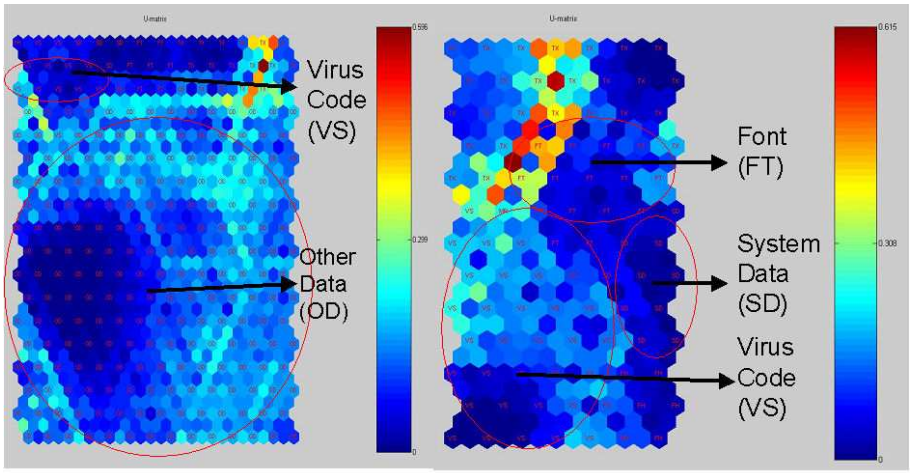

FIGURE 6.16: SOM distribution of Word97 file infected by Macro viruses

not be discussed any further in this paper). Thus, the result indicates that we cannot deal with macro viruses in the same way we deal with parasitic viruses. Thus, the current SOM-based approach is insensitive to macro viruses.

Chapter 7

Application Cases

Security used to be an inconvenience sometimes,
but now it is a necessity all the time.
- Martina Navratilova.

Until now, this project has addressed packet evaluation methods and their application of technology. In this chapter, those methods were applied to Janus, the firewall system. The details of the project were mentioned in previous chapters, whilst this chapter focuses on how Janus works based on the computational methods employed.

7.1 Janus Firewall System

7.1.1 Background of Packet Filtering & Packet Classification

The primary aspect of packet filtering is the issue of packet classification. It operates by identifying a policy by comparing the protocol header fields of a packet with a filter specification. Packet classification has been the subject of much study in recent time, for instance, [Lakshman and Stiliadis(1998), Gupta and McKeown(1999a), Feldmann and Muthukrishnan(2000)]. The reason being that the ability to classify packets plays a central role in routing and in the Differentiated Services (diffserv) [1] architecture [F. Baker and Smith(2002)]. However, the requirements to the packet classification scheme may be quite different from one application to another. One example is routing on the Internet, where the classifier is used for choosing an interface based

[1] Differentiated services enhancements to the Internet protocol are intended to enable scalable service discrimination in the Internet without the need for per-flow state and signalling at every hop. A variety of services may be built from a small, well-defined set of building blocks, which are deployed in network nodes. The services may be either end-to-end or intra-domain; they include both those that can satisfy quantitative performance requirements (e.g., peak bandwidth) and those based on relative performance (e.g., "class" differentiation).

on a routing table. Here the classification only uses one or two of the address fields in the packet header to determine route, where a firewall may classify packets based on any number of packet header fields in TCP and/or IP. A related example is whether the classification algorithms should support dynamic updates of the specification or not. This is, for instance, the case with dynamic routing. Firewalls, on the other hand, use specifications that are more static. A final difference may be the option to use dedicated hardware or not. Given these differences, common performance measures of packet classification algorithms still remain. This includes classification time, space complexity, and performance of the optimisation phase. Often worst case complexities are given in along with empirical measurements.

Packet processing has something in common, they have a set of rules and they match incoming packets to the rules to find which rules match the packet. This common function is usually called packet classification. Figure 7.1 illustrates the conceptual model of a packet classification. Packet classification function has three components; a packet, a filter database, and an algorithm to classify packets. A packet is the IP datagram to be matched with filters in the filter database. The packet has several properties that are relevant to classification; source and destination addresses, type of service, length, and so on. Filter database is a collection of filters also called rules, and each filter consists of several fields and an action. Each field of a filter is associated to a certain property of IP packet, usually to a packet header field. Each field can be in the form of single value e.g., 23, range e.g., 0-1023, prefix e.g., 203.178.143/24, or wildcard i.e., matches all values. A packet property matches to its corresponding filter field if the value of that property is contained within the value range of the filter field. A packet matches to a filter if all properties of the packet match to the corresponding fields of the filter. When this happens, packet is classified into the flow defined by the filter, and will be treated according to the action of the filter. Packet classification algorithm performs the matching between IP packet and the filters to find which filter matches the packet.

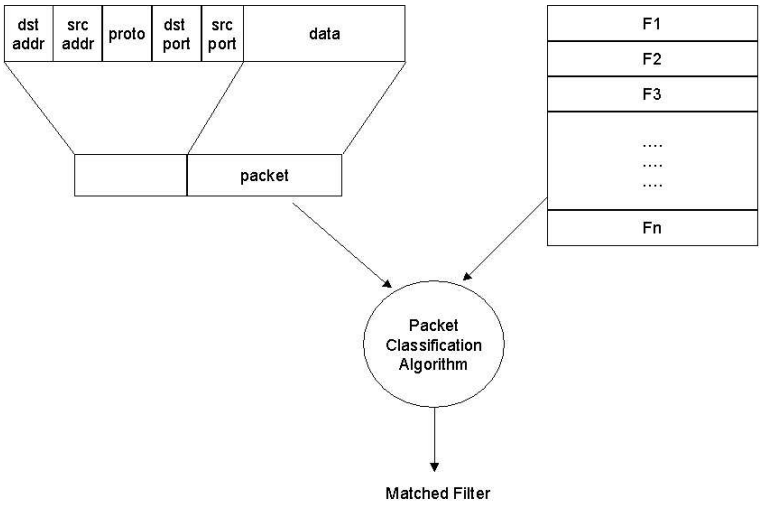

FIGURE 7.1: Conceptual model of packet classification

Packet classification can be categorized as layer-3 or layer-4 classification, depending on the fields of filters. Filters of layer 3 classification uses IP address fields or other packet header fields that are relevant for processing at the Internet layer of TCP/IP protocol stack. If classification includes other fields that are only relevant to UDP and TCP, and then it is called layer-4 classification.

Firewalls in general do layer-4 classification since they tend to use port fields for their filter databases. Firewalls are used not only to prevent or allow traffic from and to certain hosts, but also from and to certain hosts using certain applications, indicated from the protocol type and port fields. Firewall database obtained in packet classification field are less than 2000 filters [Gupta and McKeown(1999b), V. Srinivasan and Varghese(1999), Venkatachary Srinivasan and Waldvogel(1998)]. Further details of related work in packet filtering & classification is in Appendix B.

7.1.1.1 Packet Filters

Packet filters work by dropping packets based on their source or destination addresses and/or ports. In general, no context is kept; decisions are made only from the contents of the current packet. Depending on the type of router, filtering may be done at input time, at output time, or both. The administrator makes a list of the acceptable machines and services and a stop-list of unacceptable machines or services. It is easy to permit or deny access at the host or network level with a packet-filter.

TABLE 7.1: An example of a CISCO router access list.
Note. The fourth rule is never matched because of the second rule.

order	rules
1	permit udp any host 10.0.0.1 eq 53
2	deny udp any host 10.0.0.2
3	permit udp any 10.0.0.0 0.0.0.255 eq 123
4	permit udp any host 10.0.0.2 eq 177
5	deny ip any any

Simple packet-filters usually use simple ordered lists of rules. An example of a CISCO router access list is shown in Table 7.1. When a packet is received, the list is scanned from the start to the end, and the action, either “permit” or “deny”, associated with the first match is taken. If a packet does not match any of the rules, the default action is “deny”. Often a “deny all” rule is included at the end of the list to make it easier to verify that a list has not been truncated. Separate lists can be specified for each network interface.

The rules can use the following fields from the IP protocol header: next level protocol, e.g., TCP or UDP, source and destination IP addresses, type-of-service, and precedence.

In addition, some fields for upper level protocols, such as TCP and UDP port numbers can be used. A more complete discussion of the syntax of the rules used by CISCO routers is in [CISCO(2002), Hundley and Held(2000)]. Since the first matching rule is always used, it is very easy to make mistakes when writing access lists, especially when the lists are long; several hundred rules are not uncommon. For instance, the fourth rule in Table 7.1 is never matched because the packets are stopped at the second rule.

7.1.2 Process of Janus System

It is a well-known fact that all protocol stacks have security holes; these can cause vulnerabilities in firewall systems as well. However, Janus does not require a TCP/IP stack. In order to achieve this on a normal Unix computer, it was necessary to recompile the kernel with support for network cards but without any TCP/IP networking support.

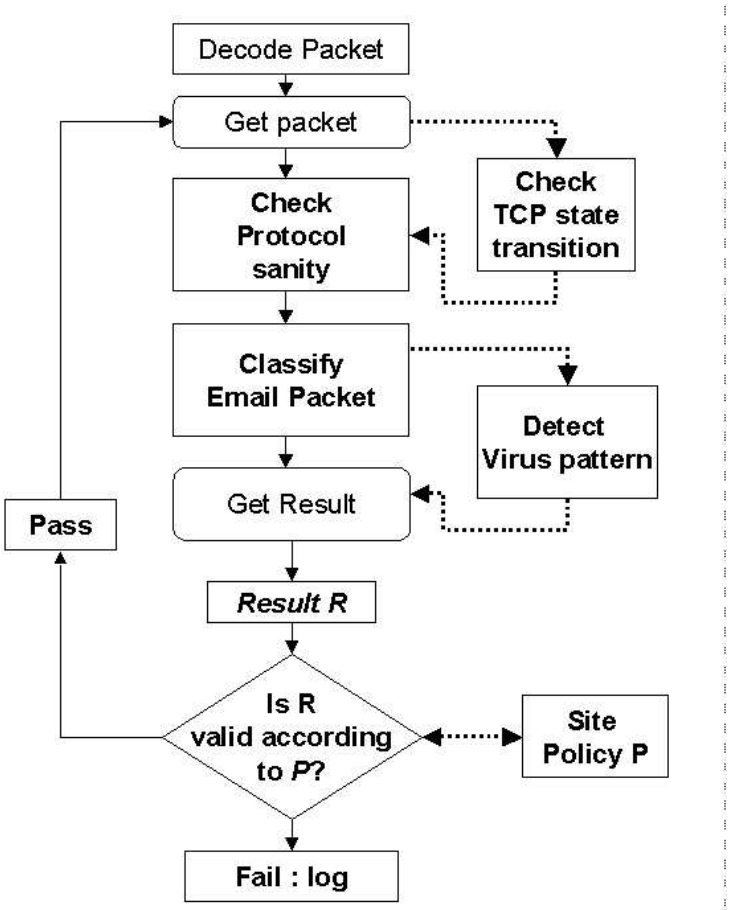

FIGURE 7.2: Work flow to make a decision in the adaptive detection model.

Figure 7.2 represents the process flow of the Janus system. Decoded packets are inspected in order to validate protocols, and are checked for protocol sanity, e.g. check header, check sender, check recipient, and check attachment. Meanwhile, TCP packets are verified using the TCP verification model. Then among TCP packets, the UBE/Virus

mail checker classifies mail packets. If mail packets' attachments contain virus possibly, virus detection alerts it. All processing parts here are predefined in previous chapters (details are found in all previous chapters); they are now applied to the real implementation. In the last step, the policy relative part needs to be updateable or modifiable. According to a site policy, the interpreter will decide whether to drop a packet or let it pass. Therefore, most policy parts are editable through rules in Janus. Through these, malicious logic, which causes the site policy to be violated, can be detected. The process flow was implemented as in Algorithm 1.

Algorithm 1 process of Janus system

```
let allPackets = {pkt | pkt ∈ ipPackets where decodedPackets(pkt) }
let receivedPackets = {rvp | rvp ∈ allPackets where decodedPackets(pkt) }
let mailPackets = {m | m ∈ TCPPackets where decodedPackets(pkt) }
let policy = {p | p ∈ accessRules & p ∈ messageRules where messagePattern(p) }
  for all pkt ∈ allPackets do
    do in parallel
      for all rvp ∈ receivedPackets do
        SanityCheck(rvp)
      end for
      choose pkt ∈ allPackets where TCPSanityCheck(pkt) do
        TCPTransactionVerification(pkt)
        choose mail ∈ mailPackets where EmailClassifier(mail) do
          VirusDetect(mail.attachment)
        end choose
      end choose
      PacketFilter(pkt, policy)
    enddo
  end for
```

7.1.3 Placement of the Janus system

The Janus firewall has been designed to protect any server system from any internal or outside network area. Like in Figure 7.3, the Janus system is placed inside the university firewalls with a separated subnetwork routing setup on an Internet connection. This causes the Janus system to act as a firewall/router on the subnetwork since all traffic travels through that system.

There are three types of placement possible: in a single server, in a multi-server, or in a grouped server environment. Current IP settings and working environment with the Janus box are in a single server environment as in Figure 7.4. The Janus packet-filter drops, and lets pass packets according to the rules defined in a config file (janus.config) and a rule file (janus.rules). For each packet inspected by the filter, the set of rules is evaluated from top to bottom, and the last matching rule decides what action is performed.

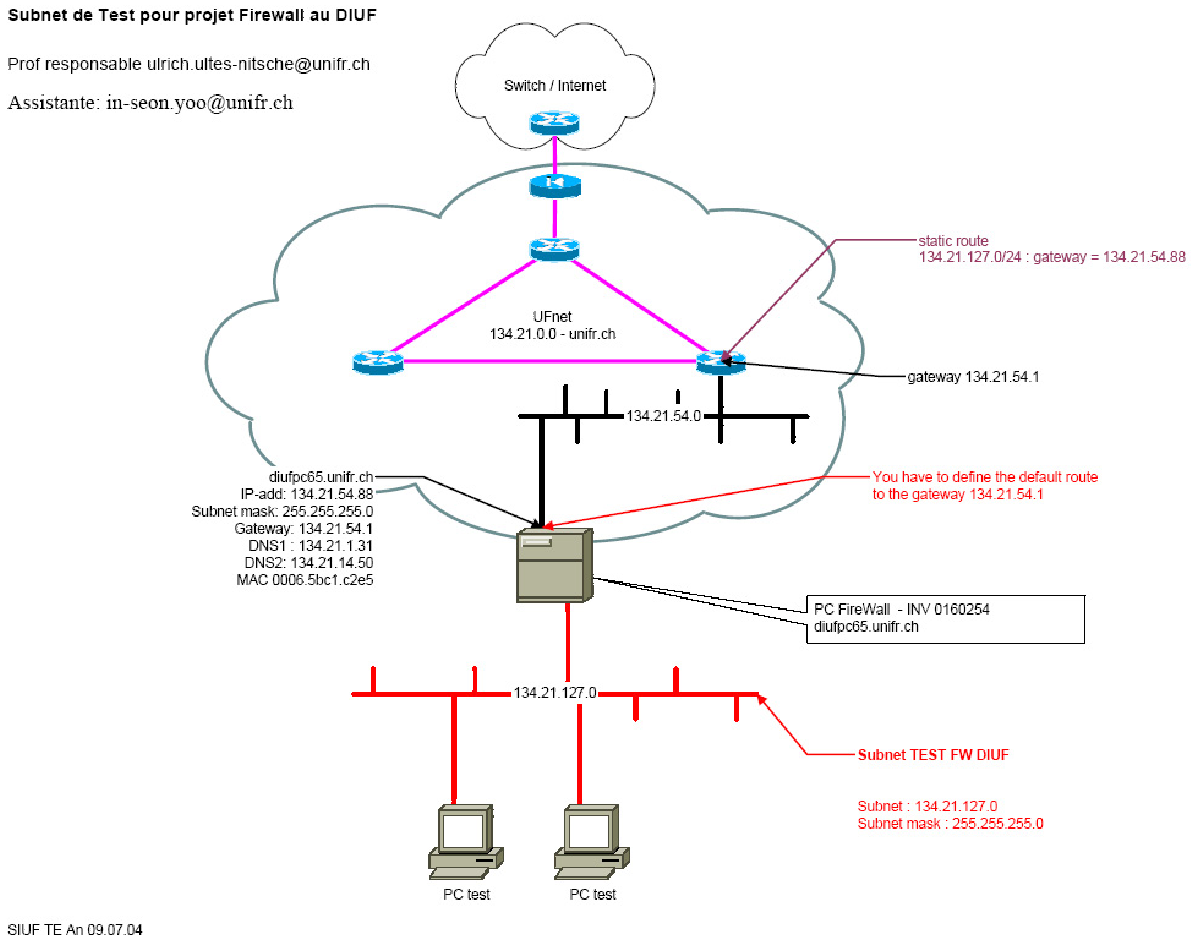

FIGURE 7.3: Janus Network Configuration

- **In a single server environment with the use of a crossover cable**. In this setup, the Janus system is attached to the network where the server used to be and the server is connected to the second network card in the Janus system via a crossover cable. This is the current Janus system configuration (Figure 7.4).

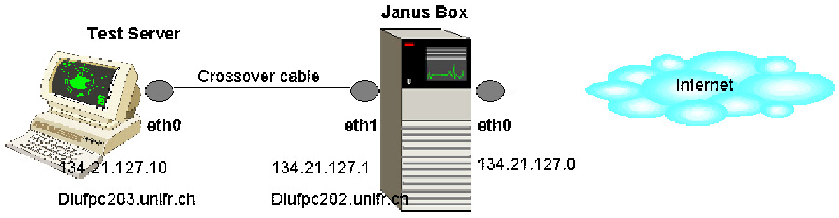

FIGURE 7.4: In a single server environment with the use of a crossover cable

- **In a multi-server environment**. In this setup, the Janus system is attached to the network where the servers used to be and the servers are either connected via a hub to the second Janus network card or multiple network cards can be used in the Janus system as in Figure 7.5.

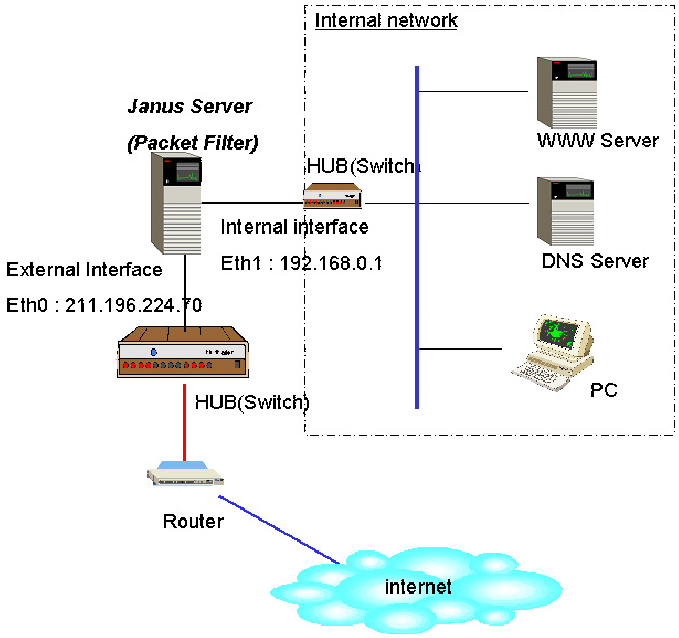

FIGURE 7.5: In a multi-server environment

- **In a grouped server environment**. Where servers are grouped together by type of services provided, separate Janus boxes can be used for each group as in Figure 7.6. In this configuration, rules are optimised by separating them into multiple Janus systems to increase performance and by grouping servers according to the services they offer to match the rules.

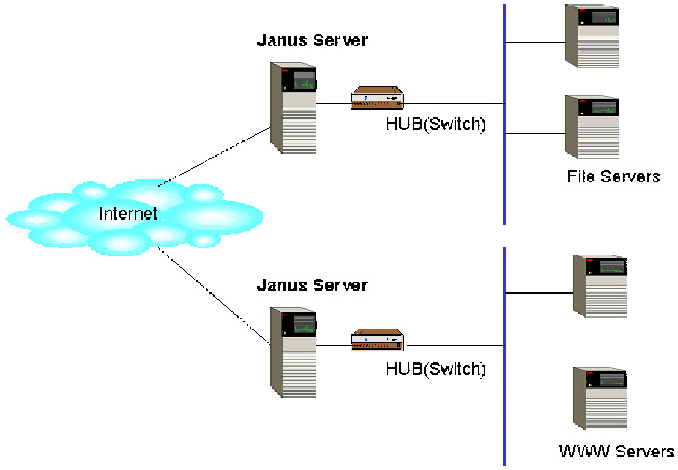

FIGURE 7.6: In a grouped server environment

7.1.4 Packet-filter & Classifier

A packet-filter is a multi-interfaced device that applies a set of rules to filter IP packets passing through its interfaces. Each interface on the device represents a connection to a network. For instance, a company protecting their internal LAN from the Internet

would need a packet-filter with two interfaces, one to the local network and another to the external network to which the local network connects such as the company's Internet service provider. Filtering on a packet-filter may occur in several places. In theory, packets can be filtered in both inbound and outbound directions on all the interfaces of a packet-filter, although this flexibility may differ between various implementations of packet-filters [Cheswick and Bellovin(1994)]. Thus, the rules of a packet-filter are organised into a set of access lists and each list is then applied to a particular interface, in a specified direction.

The main idea of packet-filter rule searching is to combine all the rules into a single data structure that can be searched once to find the first matching rule. The aim of Janus' packet-filter rule-format is to find an internal representation for access lists capable of providing fast lookup with reasonable memory requirements. Since access lists are consulted frequently for each arriving packet, the ability to perform fast lookups is the most important factor. This justifies using a potentially large amount of effort initially to create an efficient internal representation. Janus rule set consists of packet header field and message field. Since a firewall has to prevent secure systems from network attacks, Janus must cover the known network attacks by attack signatures. So, the message field can be used in this way, as well as for declaring mail attached file extensions. However, message field information needs to be chosen carefully.

7.1.4.1 Access Lists

An access list describes the security policy of the packet-filter at the point at which it is applied. Although the syntax of access lists may differ from one implementation of a packet-filter to another, their flexibility and semantics remain fairly constant. Each rule in an access list consists of a selection criterion and an action field. The action field specifies what should happen if the packet meets the selection criterion. Packet filters usually offer the two actions accept and drop. Accept means the packet should be allowed to continue towards its destination IP address whereas drop means that the packet should be discarded.

The selection criterion of a rule is a Boolean expression involving values of fields available in the packet headers. The number of fields that may be specified for filtering is known as the dimension of a filter. In theory, any information available in the packet headers can be used as filtering criteria. In practice, the common filtering fields include [Vos and Konijnenberg(1996)]:

- Source and destination IP addresses. These addresses can be found in the IP header of every packet. Traffic restriction is facilitated by allowing the administrator to specify the set of well-known hosts or networks that may use the system.

TABLE 7.2: Example of Janus Rule

```
<rule>
ip proto(UDP, ICMP, IGMP, 13-15)
action=block
</rule>

<rule>
ip dst(134.21.0.0/16, 134.21.14.50-134.21.14.59)
tcp port(21-25)
action=block
</rule>
# incoming packet towards MailServer includes executable file,
# then check for virus detection.
<rule>
ip dst(MailServers)
tcp dst(25)
tcp nocase(.exe, .pif, .com)
action=check
</rule>

# incoming packet towards Webserver includes executable file, then drop
<rule>
ip dst(WebServers)
tcp dst(80)
tcp nocase(.exe, .vbs, .com)
action=test
</rule>
```

- Transport protocol, like TCP, UDP, or ICMP, which is also found in the IP header.
- IP options, like the source route option, which is often considered dangerous for network security.
- Source and destination port numbers, associated with TCP and UDP. These can be found in the TCP and UDP headers respectively. Filters using port numbers can restrict the network traffic to a limited set of services that are associated with well-known ports.
- TCP flags, such as the ACK and SYN flags. These flags indicate whether a packet is initiating a new connection and can be used to restrict TCP connections from being initiated in certain directions.
- ICMP message types. These are found in the ICMP header. Filtering on ICMP types allows the administrator to restrict ICMP traffic to a limited set of message types.

The classification for each packet, i.e., whether it should be accepted or discarded, is given by the action field of the first rule whose selection criterion matches the packet.

Each rule of an access list describes the condition, based on the values of fields within the packet header, that a packet must meet in order to have the corresponding action effected. Thus, the condition of a rule is simply a logical expression involving certain fields of the packet header to be accepted, a packet must satisfy this expression if the rule's action is accept, and not satisfy the expression if the rule's action is reject. Current Janus rules look like Table 7.2. If no rules match, the default rule is applied. The default rule is determined by the security policy and is usually set to reject all packets, since this is the preferred security stance [R.L.Ziegler(2000)]. In the Janus rule file, "action" defines the action to perform when the rule matches. Actions are defined in the Janus config file.

7.1.4.2 Address Notation

Although the syntax of access lists differs from system to system, in terms of network addresses and masks, two conventions are used.

- base address/bit count notation: The expression 134.21.54.0/24 represents the block of IP addresses in the range 134.21.54.0 to 134.21.54.255. In other words, the number after the slash indicates the number of bits in the address, starting at the most significant, that are significant for comparison.
- address, mask tuple: The address block 134.21.54.0/24 can be alternatively expressed with a base address of 134.21.54.0 and a mask of 0.0.0.255. Both the address and the mask consist of 32 bits, with each bit in the mask specifying whether the corresponding bit in the address is significant for comparison.

7.1.4.3 Structures For Filtering Ruleset

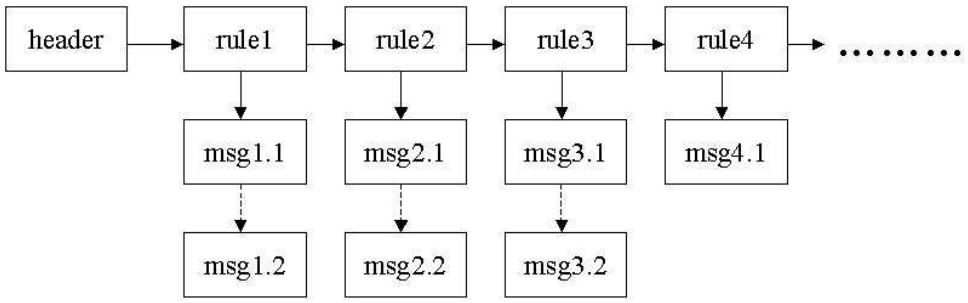

FIGURE 7.7: The linked list structure for filtering ruleset in Janus.

Janus uses a two-dimensional linked-list structure for filtering rule-set. This linked list consists of rule tree nodes and message tree nodes. A rule tree node holds many common properties that must be included in each rule, such as the source and destination addresses, source and destination ports, and protocol type such as TCP, UDP, ICMP

and so on. The message tree node holds the information for the various messages that can be added to each rule, e.g., packet content. These structures are organized into chains which can be conceptualised with the rule tree node's string from left to right as chain headers and the message tree node's hanging down from the individual rule tree nodes to which each is associated as illustrated in Figure 7.7.

When packets are being examined against a given rule set, the packet is first compared along the rule tree node list from left to right until the packet matches a particular rule tree node. Only if such a match occurs is the packet then compared down the message tree node list of the matching rule tree node. If any of the message checks fails, the packet is then checked against the next message tree node in the list. If a content check is required, Janus uses a Boyer-Moore pattern matching algorithm to check the content string held in the message tree node against the entire packet payload. If no match exists, Janus will proceed to the next message tree node in the list, which could have all options identical to the previous message tree node except for a slightly different content string.

7.1.4.4 Message Pattern Matching Algorithm

Algorithm 2 Boyer-Moore message pattern matching

```
if Output = undef or Index ≠ undef then
  if Index > Length(packetPayload) - Length(msgPattern) then
    Index := undef
  else if CharAt(msgPattern, Offset) = CharAt(packetPayload, Index + Offset)
  then
    if Offset = 0 then
      Output := Index
    else
      Offset := Offset - 1
    end if
  else
    Index := Index + Skip(Offset, CharAt(packetPayload, Index + Offset))
    Offset := Length(msgPattern) - 1
  end if
end if
```

This message pattern matching has two purposes: matching file extensions for mail attachment file, and possible attack message patterns. For message pattern matching, the Boyer-Moore algorithm was applied. Boyer-Moore [Boyer and Moore(1977)] is a quite fast pattern matching algorithm in practice. It uses heuristics to reduce the number of comparisons needed to determine if a given text string matches a particular pattern, i.e., it uses knowledge of the keyword to search for to skip over unnecessary comparisons against the text being searched. The algorithm typically aligns the text and the keyword to search for so that the keyword can be checked from left to right along the text string beginning with the last character of the keyword and ending with the first.

We will define a nullary function *Index* that holds the current index that we are testing. We will assert that its initial value is 0. If we determine that the current value of *Index* is not the start of a match, we simply increment *Index*. If we find no matches anywhere in the string, we update *Index* to undef. The algorithm is finished if a match is found, in which case *Output* is updated to some value, or if not match is found anywhere, in which case *Index* is updated to undef. We will define another nullary function *Offset*, which will be used in the character-by-character comparison. The message pattern at index *Offset* will be compared with the packet payload. A new function *CharAt* takes a string and an integer and returns the character at that position in the string. In the Boyer-Moore algorithm, the message pattern is compared against the packet payload from right to left. The message pattern still advances along the packet payload in a left to right fashion. Boyer-Moore uses the rule 'skip' over certain characters as well. If the character currently being scanned in the packet payload does not appear at all in the message pattern, we know that nothing to the left of this current character can be part of a match. So, we can move the message pattern all the way to the right of the current character. It is not the offset value that counts; rather it is the current character in the packet payload. The Boyer-Moore algorithm is presented in Algorithm 2.

7.1.5 Packet Verifier

Packet Verifier consists of *Sanity Checker* and a TCP verification model. To cover other protocol aspects apart from TCP state specification, there is Sanity Checker. This performs layer 3 and layer 4 sanity checks. These include verifying packet size, checking UDP and TCP header lengths, dropping IP options and verifying the TCP flags to ensure that packets have not been manually crafted by a malicious user, and that all packet parameters are correct. This validation is based on the protocol requirements in Section 4.1. For instance, an IP header length should always be greater than or equal to the minimal Internet header length (20 octets) and a packet's total length should always be greater than its header length. IP address checks are also important since land attacks [Fyodor(1997)] use the same IP address for source and destination. According to the TCP standard [Postel(1981c)], neither the source nor the destination TCP port number can be zero, and TCP flags, e.g. URG and PSH flags, can be used only when a packet carries data. Thus, for instance, combinations of SYN and URG or SYN and PSH become invalid. In addition, any combination of more than one of the SYN, RST, and FIN flags is also invalid. Through the requirement in Section 4.1, the implementation part of Sanity Checker procedure (function SanityCheck in Algorithm 1) is presented in Algorithm 3.

Sanity Checker examines every packet within a 10 second window, and at the end of each window, it will record any malicious activity it sees using syslog. Sanity Checker assumes any TCP packet other than a RST may be used to scan for services. If packets

Algorithm 3 SanityCheck procedure

```
let receivedPackets = {pkt | pkt ∈ ipPackets where decodedPackets(pkt) }
  for all pkt ∈ receivedPackets do
    if CheckAddress(pkt.dstaddr, pkt.srcaddr) ≠ true then
      BlockPkt(pkt)
    else
      ipProtocol := getPktIP(pkt)
      if ipProtocol = TCP then
        TCPSanityCheck(pkt)
      else if ipProtocol = UDP then
        UDPSanityCheck(pkt)
      else if ipProtocol = ICMP then
        ICMPSanityCheck(pkt)
      else
        Skip
      end if
    end if
  end for
```

of any type are received by more than 7 different ports within the window, an event is logged. The same criteria are used for UDP scans. Any SYN packets with source and destination address and ports being the same are identified as land attacks. If more than 5 ICMP ECHO REPLIES are seen within the window, Sanity Checker assumes it may be a Smurf attack [CERT(1998)]. Sanity Checker also assumes that any fragmented ICMP packet is bad, so this catches attacks such as the ping of death. Sanity Checker check any TCP fragment whether it has non-zero offset or not, so this catches fragment attack. To make the certainty higher, Sanity Checker cooperates with the TCP verification model to check three-way handshake and a SYN flood event.

The simplified TCP verification model is depicted in Figure 7.8 and the associated TCP state table with action is presented in Table 7.3 to monitor and control TCP state transitions as was presented in Chapter 4. Janus needs to trace down the behaviour represented by the remaining TCP verification model. Janus should take all kinds of TCP behaviour into account. With the TCP verification model, Janus can avoid blocking packets such that a TCP session can hang, and makes the window of opportunities for abuse as small as possible. Abuse is defined here as sending malicious data that will be accepted as valid data or sending malicious ACK's that will be accepted as valid ACK's. In addition, Janus should minimize the amount of blocked packets that belong to valid sessions. Through the verification model (Figure 7.8), the implementation part of the TCP state verification procedure (function TCPTransitionVerification in Algorithm 1) is presented in Algorithm 4.

Algorithm 4 TCP Transaction verification procedure

```
let pktHeader = {ph | ph ∈ allTCPPackets where ChooseTCPHeader(pkt) }
let tcpId = Hash (pkt.srcAddr, pkt.srcPort, pkt.dstAddr, pkt.dstPort)
  do in parallel
    rcvFlags := CheckRcvFlags(tcpId)
    sntFlags := CheckSntFlags(tcpId)
    choose tcpId ∈ pktHeader where CheckState(tcpId) do
      if UnexpectedFlags(tcpId) then
        UpdateState(tcpId, CLOSED)
      end if
    end choose
    choose tcpId ∈ pktHeader where CheckStateListen(tcpId) do
      if rcvFlags = SYN then
        UpdateState(tcpId, SYN_RCVD)
      end if
    end choose
    choose tcpId ∈ pktHeader where CheckStateSynRcvd(tcpId) do
      if rcvFlags = RST then
        UpdateState(tcpId, LISTEN)
      else if sntFlags = SYNACK then
        UpdateState(tcpId, ACK_WAIT)
      end if
    end choose
    choose tcpId ∈ pktHeader where CheckStateAckWait(tcpId) do
      if rcvFlags = ACK then
        UpdateState(tcpId, ESTABLISHED)
      else if rcvFlags = ACKFIN then
        UpdateState(tcpId, CLOSE_WAIT_2)
      else if rcvFlags = FIN & sntFlags = ACK then
        UpdateState(tcpId, CLOSING)
      else if sentFlags = RST or timeout(tcpId) then
        UpdateState(tcpId, CLOSED)
      end if
    end choose
    choose tcpId ∈ pktHeader where CheckStateClosing(tcpId) do
      if rcvFlags = ACK then
        UpdateState(tcpId, CLOSE_WAIT_1)
      end if
    end choose
    choose tcpId ∈ pktHeader where CheckStateCloseWait1(tcpId) do
      if sntFlags = ACK then
        UpdateState(tcpId, CLOSED)
      end if
    end choose
    choose tcpId ∈ pktHeader where CheckStateEstablished(tcpId) do
      if rcvFlags = FIN & sntFlag = ACK then
        UpdateState(tcpId, CLOSE_WAIT_2)
      else if rcvFlags = RST or rcvFlags = SYN or timeout(tcpId) then
        UpdateState(tcpId, CLOSED)
      end if
    end choose
    choose tcpId ∈ pktHeader where CheckStateCloseWait2(tcpId) do
      if rcvFlags = ACK or timeout(tcpId) then
        UpdateState(tcpId, CLOSED)
      end if
    end choose
  enddo
```

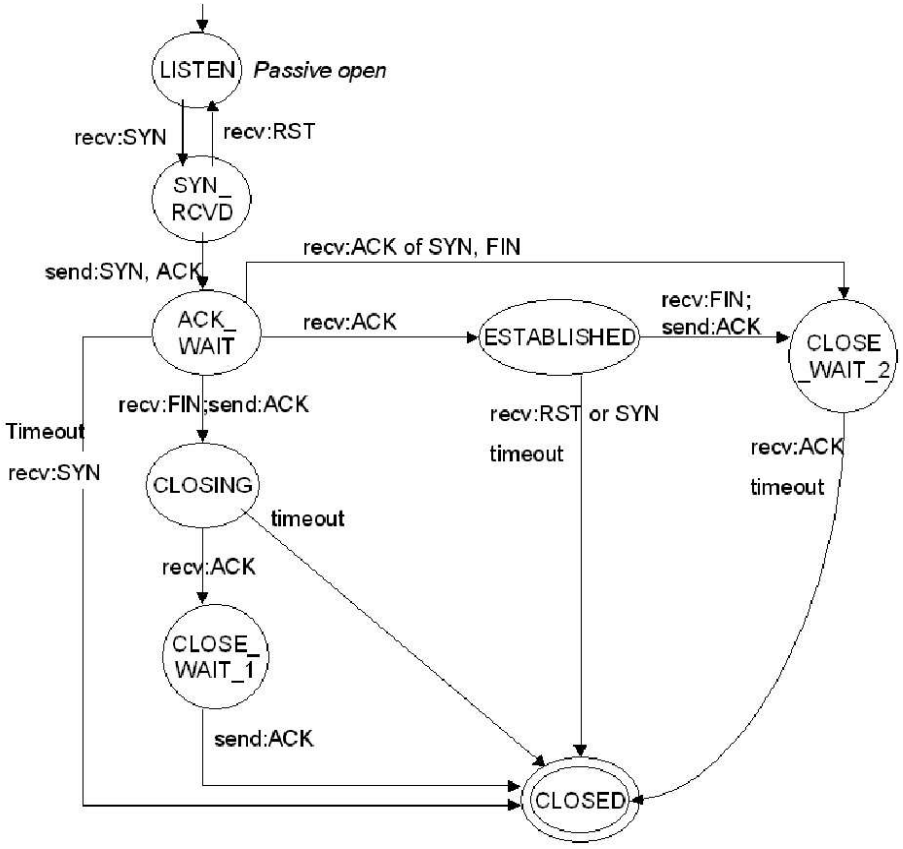

FIGURE 7.8: TCP Verification Model

TABLE 7.3: TCP states table

Current State	New State	Action
LISTEN	SYN_RCVD	recv: SYN
SYN_RCVD	LISTEN	recv: RST
SYN_RCVD	ACK_WAIT	send: SYN, ACK
ACK_WAIT	ESTABLISHED	recv: ACK
ACK_WAIT	CLOSE_WAIT_2	recv: ACK of SYN, FIN
ACK_WAIT	CLOSED	timeout (60 sec) or recv: SYN
ACK_WAIT	CLOSING	recv: FIN, send: ACK
ESTABLISHED	CLOSE_WAIT_2	recv: FIN, send: ACK
ESTABLISHED	CLOSED	recv: RST or SYN, or timeout (30 min)
CLOSING	CLOSED	timeout (10 sec)
CLOSING	CLOSE_WAIT_2	recv: ACK
CLOSE_WAIT_2	CLOSED	recv: ACK, or timeout (10 sec)

7.1.6 Email Classifier

In Chapter 5, email classification was described. With the result of classification, an email classifier is implemented as presented by Algorithm 5.

Here, most parts are predefined with the result classification information of Chapter 5. For example, the abnormal mail classifier is denoted by t, UBEs by u and Email

Algorithm 5 process for email classifier

```
let receivedMailPackets = {m | m ∈ mailPackets where decodedPackets(m)}
  for all mail ∈ receivedMailPackets do
    do in parallel
      fr := CheckSender(mail)
      to := CheckRecipient(mail)
      h := CheckHeader(mail)
      ef := CheckExecutable(mail)
    enddo
    do in parallel
      UBEflag := AlertOfUBE(fr, to)
      Virusflag := AlertOfVirus(fr, to, h, ef)
    enddo
    if  UBEflag = true & Virusflag = true then
      mailAlert := true
    else if UBEflag = true & Virusflag = false then
      ubeAlert := true
    else if UBEflag = false & Virusflag = true then
      virusAlert := true
    else
      mailAlert := false
      ubeAlert := false
      virusAlert := false
    end if
  end for
```

viruses by v. Therefore the abnormal mail classifier t is true if either u or v are true, which means that this classifier can say whether or not a mail is abnormal by UBE or email-virus data in the raw packets detected.

$$t = u + v.^{2}$$

The UBE classifier u is true if either a sender Fr or a recipient To are false, which means that this classifier can say whether a mail is an UBE by identifying that either the sender part or the recipient part are malformed.

$$u = \neg(Fr \wedge To)$$

The classifier decides whether mail contains an email virus by the following facts; although the sender part is correct, a recipient part and a header part are wrong and there is an attachment in the mail, or although the recipient part is correct, a sender part and a header part are wrong and there is an attachment in the mail, or a sender part and a recipient part are wrong even though the header part is ok and there is an attachment,

[2] "+" represents here the Boolean OR operation.

or a sender part, a recipient part and a header part are all wrong whether there is an attachment or not. For email virus classification v,

$$v = (Fr \wedge \neg To \wedge \neg H \wedge EF) \vee (\neg Fr \wedge To \wedge \neg H \wedge EF) \vee (\neg Fr \wedge \neg To \wedge H \wedge EF) \vee (\neg Fr \wedge \neg To \wedge \neg H)$$

UBE and email virus information is implemented in functions AlertOfUBE and AlertOfVirus. The process of email classification is implemented by Algorithm 5.

7.2 Janus VirusDetector

Prior work presented in Chapter 6 described an approach to visualize virus patterns using Self-Organizing Maps (SOMs) [Kohonen(1995)]. The SOM visualization of virus-infected files proved that the virus detection approach without prior knowledge of virus signatures using SOM made sense. In this section, a virus detection program, *VirusDetector*, is presented which has been developed for determining whether a file is virus-infected or not, based on the prior work using SOM.

A virus detection program, *VirusDetector*, was then developed which uses the SOM algorithms initially employed in the MATLAB visualization. However, it does not visualize the viruses anymore but solely decides whether or not the visualization had contained the so-called *virus mask* which indicates the presence of a virus.

Even though a colourful visualization was produced easily using MATLAB, *VirusDetector* does not require colourful visualization; only the Umatrix neurons' values are used for detection. Thus, *VirusDetector* uses an internal Black/White representation of the Umatrix's node values using textual information.

7.2.1 Test Data Collection

In total, 790 virus-infected files (291 Win9x files and 499 Win32 files), 80 normal (i.e. non-infected) Windows executable files, and 15 macro-virus-infected Windows Word files were tested using the SOM-based approach, including downloadable application programs such as SSHSecureShellClient-3.2.9.exe, dxwebsetup.exe, klcodec220b.exe, and already installed executable programs such as Excel.exe, Winword.exe, Acrobat.exe, servertool.exe. Those virus-infected files were detected and caught in between 1996 and 2004. Even "old" viruses are still of interest as they have several variants that appear in present days. For example, variants of the CIH/Chernobyl [CERT(1999b)] virus have appeared each year since 1998. All test data, file information and the test results are listed in Appendix C.

7.2.2 Process of Virus Detection

With the knowledge about the virus mask in SOM projections of virus-infected files, *VirusDetector* is built which, based on SOM algorithms, detects the virus mask. Apart from the colourful visualization result using MATLAB, *VirusDetector* uses a simple Black/White colour scheme.

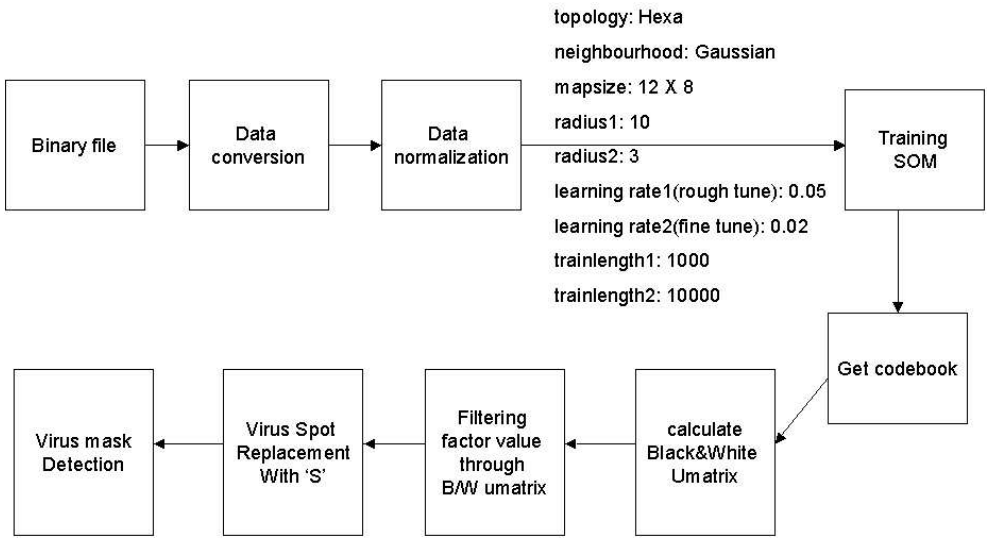

FIGURE 7.9: The process of virus detection in *VirusDetector*

Figure 7.9 shows the step-wise process in *VirusDetector*. After feeding a binary executable file to *VirusDetector*, the data is converted into integer format. Then the data is normalized using certain SOM parameters such as topology type: Hexa, neighbourhood: Gaussian, map size: 12 x 8, radius1: 10, radius2: 3, learning rate1: 0.05 for rough tune, learning rate2: 0.03 for fine tune, trainlength1: 1000, and trainlength2: 10000. Afterwards, the SOM is trained with the normalized data. Unfortunately, there is no theoretical basis for the selection of these parameters [Kohonen(1995)]. The parameter settings were determined experimentally from the visualization results using the SOM toolbox in MATLAB. Note that the graphical representation of the SOM depends on the initialisation, meaning that a virus mask might be located in a different location, or be shown in a totally undetectable form for a different initialisation. However, using the mentioned (good) parameters, the SOM projection produces the patterns which *VirusDetector* can search for and finally find the virus mask if it is present.

While the SOM is trained, it produces a codebook for storing the data. Using this codebook, *VirusDetector* calculates the Umatrix in Black/White. High values (dark SOM cells) indicate high data density, which is particularly the case for virus data. A "factor value" is used for selecting which high values are significant. According to these B/W Umatrix values, *VirusDetector* filters out the Umatrix values above the factor value and saves them. To represent the filtered-out values on the two-dimensional projection plane, the character "S" is used (each SOM cell with an assigned gray-scale value above

the factor value is represented by an "S" all other cells are blank). This produces a representation of the projection plane in form of ASCII strings. After producing the map of "S"'s, *VirusDetector* searches for the virus pattern and marks the virus mask if any is present. After identifying the virus mask, *VirusDetector* decides that the file under test is virus-infected.

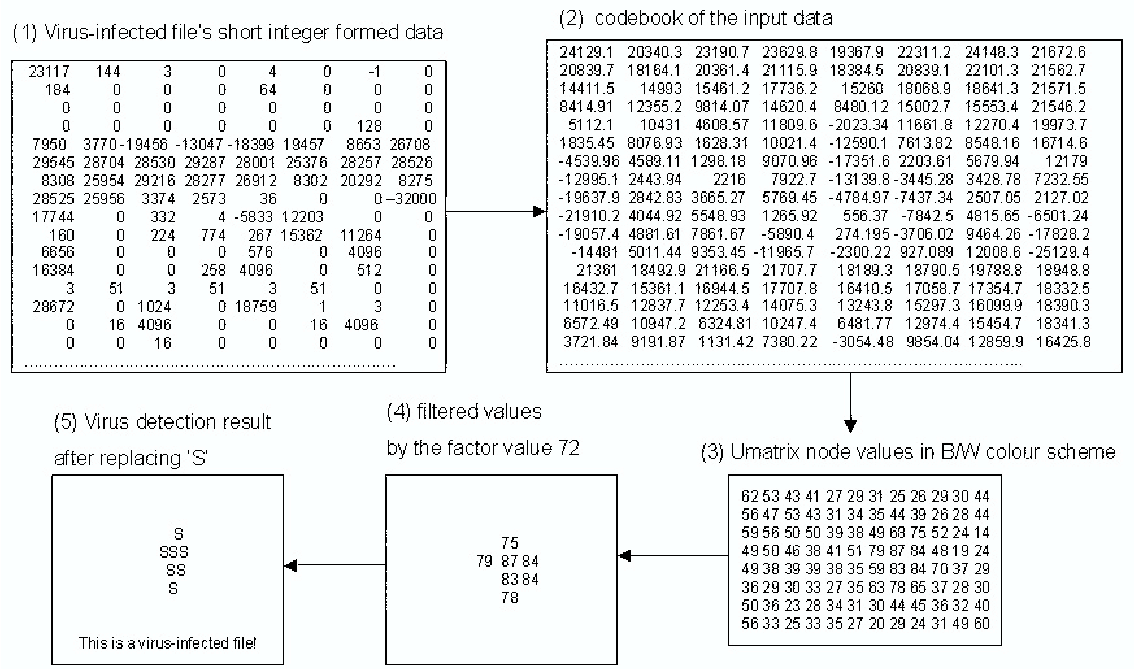

FIGURE 7.10: Virus Detection Example

Here is an example of the detection process as depicted in Figure 7.10.

1. Data buffer presents a virus-infected file's short-integer-formatted data.
2. After normalization in *VirusDetector*, the codebook of the input data is produced.
3. *VirusDetector* calculates the values of the Umatrix neurons from the codebook, assigning a grey-scale colour value to each neuron (SOM cell).
4. If the factor value is 72, which is used in *VirusDetector*, the neurons with a higher value than the factor value are selected, as presented in Figure 7.11 (a).
5. The filtered values are replaced by character "S" to create strings as presented in Figure 7.11 (b), and *VirusDetector* manipulates the strings to search for the virus pattern. In the final step, *VirusDetector* decides on the found pattern whether or not it represents a virus mask using Algorithm 6.

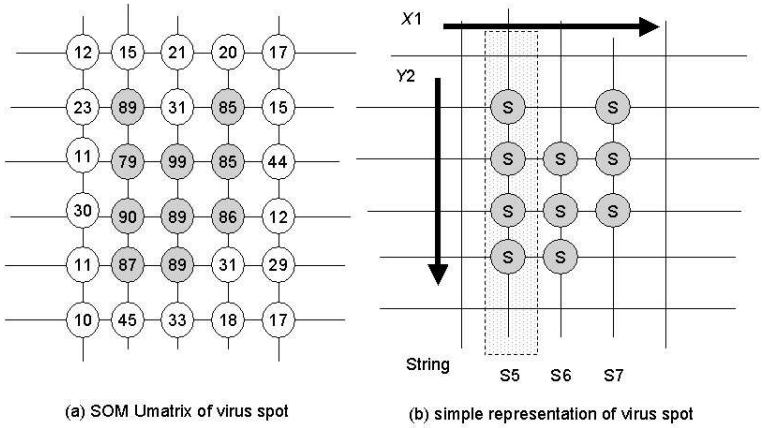

FIGURE 7.11: In B/W Umatrix, bigger values are selected and replaced by char S'.

Algorithm 6 process of searching virus mask

```
for all string ∈ givenfile do
  if search string S_i for the first occurrence of substring "SSS" then
    if search string S_i+1 for the first occurrence of substring "SS" then
      compare up to each string's size characters of string S_i to S_i+1, and vice
      versa.
      if size string is equal to or less than the other string, then
        virus mask!
      end if
    else if repeated search on string S_i for substring "SSSS" then
      if search string S_i+1 for the first occurrence of substring "SSS" then
        virus mask!
      end if
    end if
  else
    compare up to string S_i's size characters of string S_i to S_i+1,
    if S_i is equal to or less than S_i+1, then
      virus mask!
    end if
  end if
end for
```

7.2.3 Result of Virus Detection

Using *VirusDetector*, the 790 virus-infected files are tested, which are listed in Appendix C. The test set only includes virus-infected executable Windows files. Since experiments with labelling the input data showed that it could not detect macro viruses properly, macro viruses are excluded from the test set.

7.2.4 Unencrypted Parasitic Viruses

Figures 7.12, 7.13 and 7.14 are the SOM projection of the CIH virus v1.2, v1.3, v1.4 and recognition result produced by *VirusDetector* respectively. As the result shows, the virus mask is represented roughly by the strings of "S"'s. Figures 7.15 and 7.16 contains the equivalent result for Win95.Anxiety virus. Tables 7.4 and 7.5 summarize the experiments conducted on some virus-infected executables in Win9x and Win32 format respectively. Results on the recognition of non-infected executable files are presented in Table 7.6. The complete result list can be found in Appendix C.

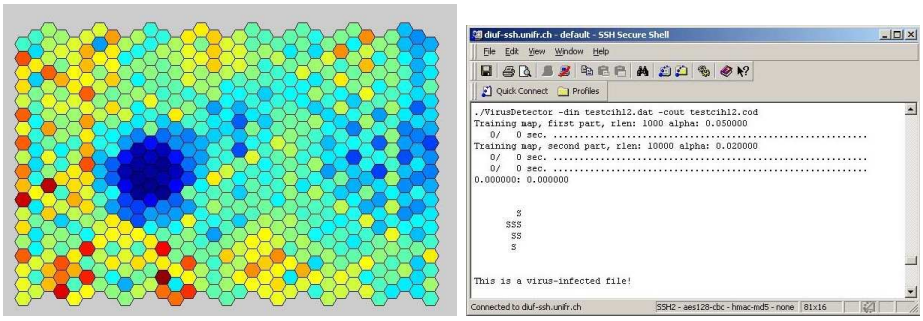

FIGURE 7.12: SOM Umatrix and detection result of CIH 1.2 virus

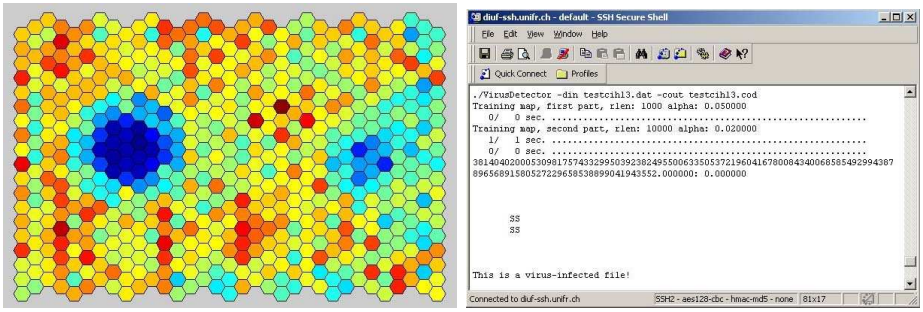

FIGURE 7.13: SOM Umatrix and detection result of CIH 1.3 virus

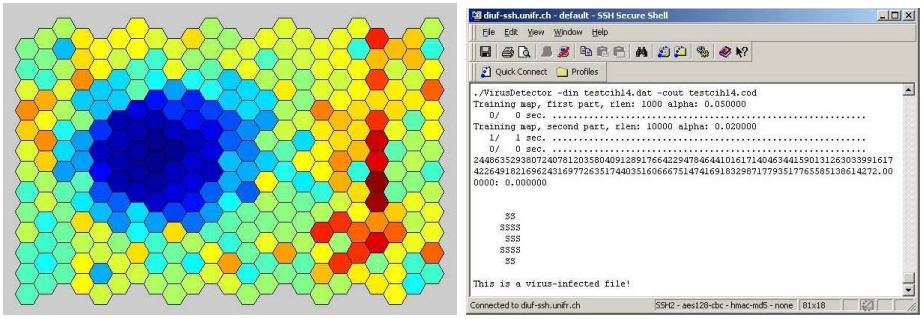

FIGURE 7.14: SOM Umatrix and detection result of CIH 1.4 virus

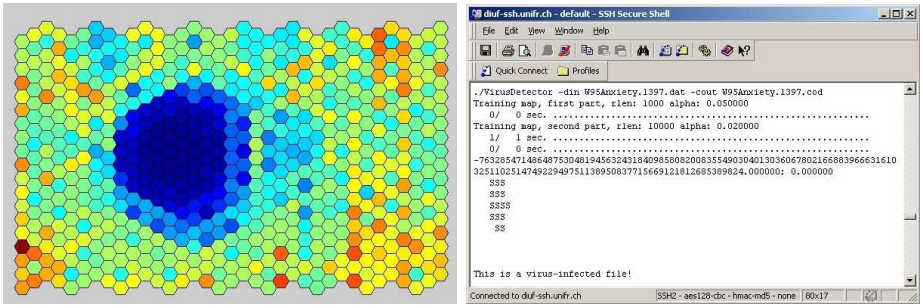

FIGURE 7.15: SOM Umatrix and detection result of W95 Anxiety.1397 virus

TABLE 7.4: Win9x Virus Detection Result by *VirusDetector*.
Win9x Virus Total Number: 291, Error number: 26.
False negative: 0.09 (approx. 9%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Altar.797	8192	6-19-99	O	Altar.884	4096	10-4-02	O
Altar.910	4096	10-24-99	O	Antic.695	7863	9-2-02	O
Anxiety.1358	152778	3-1-04	O	Anxiety.1397	49736	3-1-04	O
Anxiety.1399	8192	3-21-98	O	Anxiety.1399.b	49736	3-1-04	O
Anxiety.1422	5684	11-22-03	O	Anxiety.1451	29213	2-21-01	O
Anxiety.1486	8192	1-24-98	O	Anxiety.1517	8192	11-22-03	O
Anxiety.1586	42590	11-22-03	O	Anxiety.1596	49736	3-11-98	O
Anxiety.1823	8192	3-1-04	O	Anxiety.1823.b	6196	7-2-03	O
Anxiety.2471	8192	11-22-03	O	Apop.1086	8192	7-17-02	O
Argos.310	4096	9-2-02	O	Argos.328	4096	3-1-04	O
Argos.335	4096	9-24-02	O	Argos.402	4096	6-30-99	O
Bodgy.3230	97438	4-15-03	X	Bonk.1232	19632	11-22-03	O
Bonk.1243	9460	3-1-04	O	Boza.2220	24576	11-22-03	X
Boza.a	12408	3-17-96	O	Boza.b	7994	11-22-03	O
Boza.c	16384	3-1-04	O	Boza.d	16384	11-22-03	O
Boza.e	16384	11-22-03	O	ByteSV.Thorn.886	20480	11-22-03	O
Caw.1262	205550	11-22-03	O	Caw.1335	5943	3-1-04	O
Caw.1416	55964	11-22-03	O	Caw.1419	54667	11-22-03	X
Caw.1457	6065	11-22-03	O	Caw.1458	8192	10-5-02	O
Caw.1525	24576	11-22-03	O	Caw.1531	8192	11-2-01	O
Caw.1557	180224	12-31-00	O	Chimera.1542	35846	11-22-03	O
CIH	19536	7-8-02	O	CIH.1003.b	4608	3-1-04	O
CIH.1010.b	37394	11-22-03	O	CIH.1016	34304	9-2-02	O
CIH.1019.c	4896	3-1-04	O	CIH.1024	1553	3-1-04	O
CIH.1026	1555	3-1-04	O	CIH.1031	4096	9-2-02	O
CIH.1035	1564	3-1-04	O	CIH.1040	159744	3-1-04	O
CIH.1048	20480	9-2-02	O	CIH.1049	65536	9-2-02	O
CIH.1103	2144	11-22-03	O	CIH.1106	153088	11-27-02	O

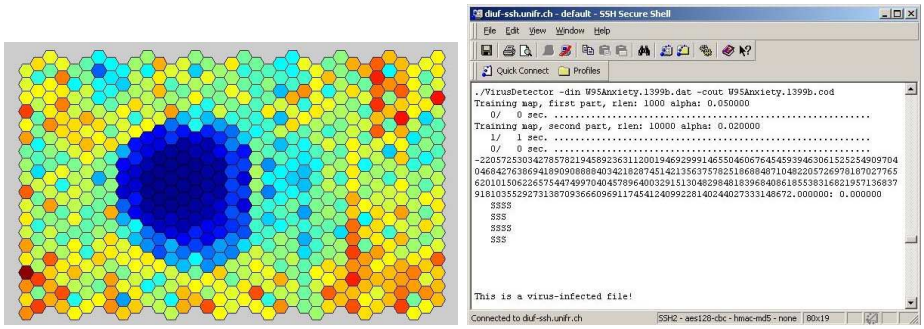

FIGURE 7.16: SOM Umatrix and result of of W95 Anxiety.1399b virus

TABLE 7.5: Win32 Virus Detection Result by *VirusDetector*.
Win32 Virus Total Number: 499, Error number: 103.
False negative: 0.2064 (approx. 21%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is Byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Adson.1559	8192	7-26-02	O	Adson.1734	20480	11-22-03	O
Aidlot	8192	10-4-04	O	Akez	32768	4-27-02	O
Aliser.7825	12288	3-6-03	O	Aliser.7897	8192	6-8-03	O
Aliser.8381	8192	6-8-03	O	Alma.2414	10606	3-1-04	O
Awfull.2376	3072	9-9-03	O	Awfull.3571	4096	3-1-04	O
Bakaver.a	24576	10-6-03	O	Banaw.2157	8192	11-22-03	O
Barum.1536	5632	8-31-02	O	Bee	24576	3-1-04	O
Beef.2110	57344	3-1-04	O	Belial.2537	8192	11-22-03	O
Belial.2609	254513	3-9-02	X	Belial.a	4096	3-10-04	O
Belial.b	4096	2-23-02	O	Belial.c	4096	11-22-03	O
Belial.d	4096	9-24-02	O	Belod.a	8192	3-12-02	O
Belod.b	8192	8-21-02	O	Belod.c	8192	3-14-02	O
Bender.1363	3584	12-31-01	O	Bika.1906	8192	12-31-01	O
BingHe	296643	10-11-02	X	Blackcat.2537	8192	9-9-03	O
Blakan.2016	8192	12-31-01	O	Blateroz	8192	9-2-02	O
Blueballs.4117	16384	11-22-03	X	Bobep	8192	8-25-03	O
Bogus.4096	38400	10-13-99	O	Bolzano.2122	36864	2-10-03	O
Bolzano.2664	15135	2-10-03	O	Bolzano.2676	15183	2-10-03	O
Bolzano.2716	13521	2-10-03	O	Bolzano.3100	15277	2-10-03	O
Bolzano.3120	15331	2-10-03	O	Bolzano.3148	15373	2-10-03	O
Bolzano.3164	15409	2-10-03	O	Bolzano.3192	15457	3-1-04	O
Bolzano.3628	16095	3-1-04	O	Bolzano.3904	16251	2-10-03	O
Bolzano.5572	28237	2-10-03	O	Butter	96665	9-2-02	X
Cabanas.a	7171	5-7-04	X	Cabanas.b	7171	3-1-04	O
Cabanas.Debug	95748	10-4-04	O	Cabanas.e	16384	7-8-03	O
Cabanas.MsgBox	39996	10-4-04	X	Cabanas.Release	49152	1-26-99	O
CabInfector	4096	3-1-04	O	Cecile	28672	12-31-01	X
Cefet.3157	7253	9-2-02	O	Cerebrus.1482	8192	3-1-04	O

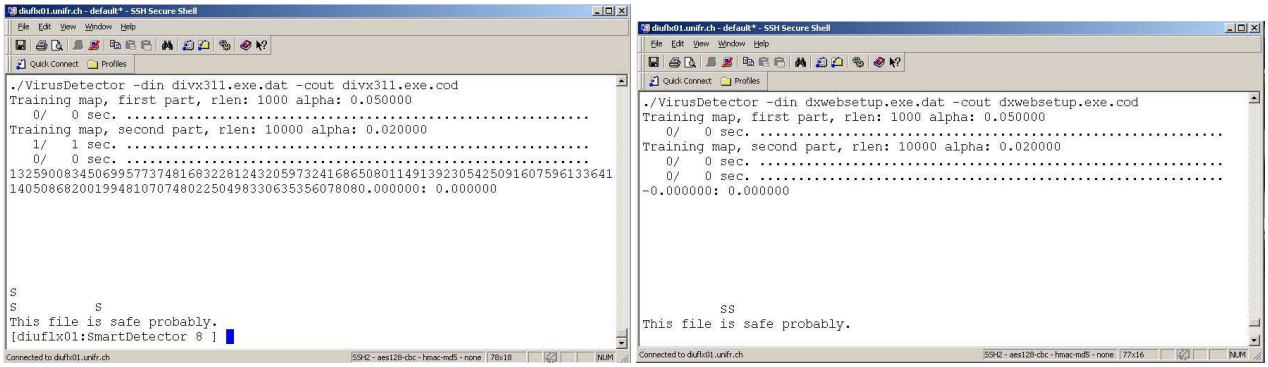

FIGURE 7.17: Virus Detection Result Scene 3: normal executable file cases.

TABLE 7.6: Normal Executable Program's Virus Check Result by *VirusDetector*
Normal Executable File's Total Number: 80, Error number: 24.
False positive: 0.3 (approx. 30%).
If the result of detection is marked X, *VirusDetector* says this file is a virus-infected file, which means incorrect detection.

Filename	Detection	Filename	Detection
dxwebsetup.exe		divx311.exe	
awsepersonal.exe		DVD2DIVXVCD_trial.exe	
paulp_en1.exe		adrenalin2.0.1.exe	
GOMPLAYER14.exe		sdvd190.exe	
csdl13.exe		HwpViewer.exe	
SSHSecureShellClient-3.2.9.exe		DivX505Bundle.exe	
klcodec220b.exe		SwansMP24a-WCP.exe	
DivXPro511GAINBundle.exe		NATEON.exe	X
WinPcap_3_1_beta_3.exe		ducp708_type3_free.exe	
wmpcdcs8.exe		Acrobat.exe	
iTunes.exe		pccmain.exe	
sicstusc.exe		Tra.exe	
acrodist.exe		java.exe	X
PCcpfw.exe		sicstus.exe	
Trialmsg.exe	X	Ad-Aware.exe	
javaw.exe	X	PCctool.exe	
splfr.exe	X	tsc.exe	
AdobeUpdateManage.exe		jpicp132.exe	X
Photoshp.exe		spmkds.exe	
conf.exe		policytool.exe	X
spmkrs.exe	X	unregaaw.exe	
CSDL.exe		kinit.exe	X
Sshclient.exe		ssh2.exe	

7.2.5 Polymorphic and Encrypted Parasitic Viruses

Among the 790 virus-infected test files, there were 30 Win9x encrypted and 15 polymorphic parasitic viruses. These are about 15% of the tested Win9x virus-infected files. In case of Win32 executables, 30 were encrypted and 50 were polymorphic. This represents again approximately 16% of all tested Win32 virus-infected files. The way *VirusDetector* checks these files is identical to testing for unencrypted, non-polymorphic viruses. The results in the encrypted or polymorphic case are quite noticeable. In case of Win9x executables, either encrypted or polymorphic viruses were detected easily by *VirusDetector* with 3% and 13% false negative rate respectively. In case of Win32 executables containing an encrypted virus, the false negative rate was lower (13%) than *VirusDetector*'s average false negative rate on the entire data set (presented in Tables 7.7 and 7.9). However, the false negative rate for Win32 executables infected with a polymorphic virus was much higher (42%) than average (16%) (see Tables 7.8 and 7.10).

TABLE 7.7: Win9x Encrypted Parasitic Virus Detection Result by *VirusDetector*.
Encrypted Win9x Virus Total Number: 30, Error number: 1, False negative: 0.03 (3%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Bumble.1736	8192	3-1-04	O	Bumble.1738	8192	11-22-03	O
Iced.1344	44032	9-6-01	O	Iced.1376	8192	11-22-03	O
Iced.1412	8192	11-22-03	O	Iced.1617	8192	3-1-04	O
Iced.2112	8192	3-1-04	O	Mad.2736.a	32768	10-4-04	O
Mad.2736.b	32768	10-4-04	O	Mad.2736.c	32768	1-7-02	O
Mad.2736.d	32768	1-7-02	O	Mad.2806	32768	1-19-98	O
Nathan.3276	7372	9-2-02	O	Nathan.3520.a	12288	3-10-04	O
Nathan.3520.b	16384	9-9-01	O	Nathan.3792	185552	10-23-99	O
Obsolete.1419	5003	3-1-04	O	PoshKill.1398	8192	4-21-01	O
PoshKill.1406	8192	3-17-03	O	PoshKill.1426	8192	8-20-01	O
PoshKill.1445.a	8192	3-10-04	O	PoshKill.1445.b	8192	11-22-03	O
Priest.1419	4096	8-9-99	O	Priest.1454	4096	3-1-04	O
Priest.1478	9728	6-10-00	X	Priest.1486	9728	10-4-01	O
Priest.1495	9728	8-18-01	O	Priest.1521	9728	3-1-04	O
Shoerec	321536	8-12-01	O	Tip.2475	10752	11-22-03	O
Voodoo.1537	61441	11-22-03	O	Werther.1224	6344	12-31-01	O

All encrypted Win9x viruses used in the tests are listed in Table 7.7, all polymorphic Win9x viruses are in Table 7.8, all encrypted Win32 viruses are in Table 7.9, and all polymorphic Win32 viruses are in Table 7.10, each attached with their size, the date when they were caught and the test result of *VirusDetector*. For information about these polymorphic and encrypted parasitic viruses, please refer to the site about Windows viruses at KASPERSKY (Metropolitan Network BBS Inc., Bern, Switzerland)

TABLE 7.8: Win9x Polymorphic Virus Detection Result by *VirusDetector*.
Win9x Polymorphic Virus Total Number: 15, Error number: 2, False negative: 0.13 (13%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Begemot	8192	3-1-04	O	Darkmil.5086	12288	3-1-04	X
Darkmil.5090	74210	7-2-03	O	Fiasko.2500.a	8192	11-22-03	O
Fiasko.2500.b	12935	3-1-04	O	Fiasko.2508	8192	3-1-04	O
Invir.7051	9728	3-1-04	O	Luna.2636	8192	4-24-02	O
Luna.2757.a	62213	3-10-04	O	Luna.2757.b	12288	1-1-80	O
Marburg.a	493789	3-10-04	O	Marburg.b	28381	11-22-03	X
Matrix.3597	35916	2-14-03	O	Merinos.1763	9216	3-1-04	O
Merinos.1849	8192	11-22-03	O				

[KASPERSKY(1994-2005)].

TABLE 7.9: Win32 Encrypted Parasitic Virus Detection Result by *VirusDetector*.
Encrypted Win32 Virus Total Number: 30, Error number: 4, False negative: 0.13 (13%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Ditto.1488	6096	3-1-04	O	Ditto.1492	12288	11-22-03	O
Ditto.1539	8192	10-1-00	O	Gloria.2820	16384	11-22-03	X
Gloria.2928	16384	3-1-04	O	Gloria.2963	12288	10-1-00	O
Idele.2104	8192	7-8-03	O	Idele.2108	8192	3-1-04	O
Idele.2160	8192	11-12-03	O	IhSix.3048	8192	11-22-03	O
Infinite.1661	8192	3-1-04	O	Levi.2961	12288	5-16-01	O
Levi.3040	7188	11-22-03	O	Levi.3090	12288	11-22-03	O
Levi.3137	35941	11-22-03	O	Levi.3205	12288	3-1-04	X
Levi.3240	16384	8-17-02	O	Levi.3244	16384	11-22-03	X
Levi.3432	16384	11-22-03	O	Mix.1852	4096	5-30-00	O
Niko.5178	65611	11-22-03	X	Santana.1104	81920	12-4-01	O
Savior.1680	8192	1-8-01	O	Savior.1696	12288	5-18-01	O
Savior.1740	12288	3-28-02	O	Savior.1828	20480	8-6-01	O
Savior.1832	12288	3-1-04	O	Savior.1904	12288	12-4-01	O
Undertaker.4887	12288	11-22-03	O	Undertaker.5036.a	12288	11-22-03	O

TABLE 7.10: Win32 Polymorphic Virus Detection Result by *VirusDetector*.
Encrypted Win32 Virus Total Number: 50, Error number: 21, False negative: 0.42 (42%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Andras.7300	14238	7-27-02	O	AOC.2044	8192	11-28-99	O
AOC.2045	8192	11-26-99	O	AOC.3657	16384	3-1-04	O
AOC.3833	16384	3-1-04	O	AOC.3860	20480	2-10-03	X
AOC.3864	20480	2-10-03	O	Champ	12288	2-10-03	O
Champ.5430	12288	10-6-02	O	Champ.5464	12288	2-10-03	O
Champ.5477	12288	10-6-02	O	Champ.5495	6144	2-10-03	X
Champ.5521	12288	2-10-03	X	Champ.5536	12288	2-10-03	X
Champ.5714	12288	2-10-03	O	Champ.5722	16384	2-10-03	X
Chop.3808	64049	8-29-01	X	Crypto	49152	3-1-04	X
Crypto.a	28672	11-22-03	X	Crypto.b	32768	11-22-03	O
Crypto.c	32768	11-22-03	O	Driller	94208	3-1-04	O
Harrier	108544	11-22-03	X	Hatred.a	16384	3-10-04	O
Hatred.d	16384	10-29-02	O	Kriz.3660	415232	7-27-02	X
Kriz.3740	764928	10-4-04	X	Kriz.3863	475136	10-4-04	X
Kriz.4029	12288	3-1-04	O	Kriz.4037	12288	8-19-01	O
Kriz.4050	479232	11-22-03	X	Kriz.4057	12288	8-19-01	X
Kriz.4075	12288	11-22-03	O	Kriz.4099	12288	11-22-03	X
Kriz.4233	8192	7-15-01	X	Kriz.4271	57344	10-4-04	O
Prizm.4428	8704	9-4-02	X	RainSong.3874	8192	11-22-03	O
RainSong.3891	61509	3-1-04	O	RainSong.3910	8192	12-5-01	X
RainSong.3956	12288	10-4-04	X	RainSong.4198	8192	3-12-02	O
RainSong.4266	12288	10-4-04	X	Thorin.11932	16384	3-1-04	O
Thorin.b	16384	3-1-04	O	Thorin.c	16384	10-23-99	O
Thorin.d	16384	7-14-99	O	Thorin.e	16384	10-23-99	O
Vampiro.7018	18432	3-1-04	X	Vampiro.a	16896	3-10-04	O

7.2.6 False positive vs. False negative in *VirusDetector*

A false positive occurs when a non-infected file is tested and the test result categorizes the file as positive (i.e. infected). In the inverse case of a not-detected-infected file, the outcome is called a false negative. The false negative rate and the false positive rate are interdependent; to decrease one is to increase the other. Therefore, it is important to decide which side to decrease and which to increase. A significant role is played by the factor value, one uses in *VirusDetector* to decide whether the value of SOM cell is significant or not (i.e. whether or not the cell is likely to present a fraction of the virus mask). To determine the threshold of decision in certain patterns, the test results of all 790 virus-infected and 80 non-infected Windows executable files were taken into account.

For different factor values, the false-positive and false-negative rates of the tests on the entire data set are presented in Figure 7.18.

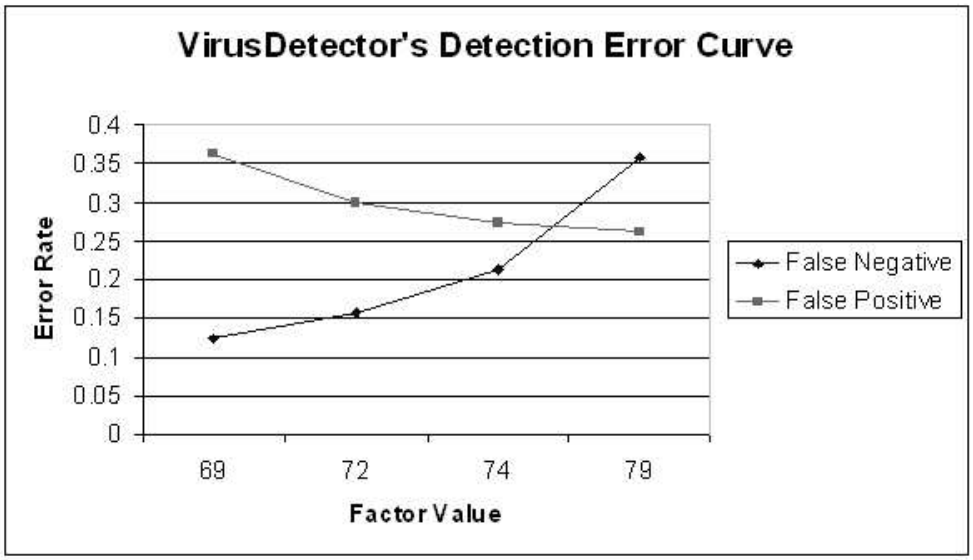

FIGURE 7.18: VirusDetector's Error Curve based on factor values

As Figure 7.18 shows, the false positives remain more or less in the same range (0.25 - 0.3) for factor values 72 and 79. On the other hand, the false-negative rate is increased significantly. So, without knowing any virus signature and anything else about a virus, *VirusDetector* can detect a virus infection with a probability of 84% and false positive rate of 30%. Some further fine-tuning might be necessary, possibly on the cost of reducing the detection capabilities (i.e. increasing the false-negative rate), in order to decrease the false positives. Using the factor value (72), all the files (790 virus-infected files and 80 non-virus-infected normal executable files) were tested. The full list of information and results on the tested files is given in the appendix. Among the 291 Win9x virus-infected files, *VirusDetector* failed to detect 26 (approximately 9%; see Table 7.4). Among the 499 Win32 virus-infected files, *VirusDetector* did not succeed in detecting 103 (approximately 21%; see Table 7.5). On the other hand, among the 80 non-infected Windows executable files, *VirusDetector* failed to pass 24 (approximately 30%; see Table 7.6).

7.2.7 Discussion of *VirusDetector*

The virus-detection approach presented in this paper exploits the detection capabilities of a self-organizing map (SOM). It basically uses structural information about the data contained in an executable file: the virus code is data injected into a formerly complete and (sort of) homogeneous structure, namely the program code. Hence the virus, even though not easily detectable by standard techniques (assuming that the virus signature is not known), is "somewhat different" from the program it infected. The SOM used in the non-standard way, is capable of just doing that: reflecting the presence of data in

an executable, which is somehow different from the rest. Whether the injected code was an encrypted parasitic or a polymorphic parasitic virus does not matter; it still differs from the rest of the program code. In SOM terminology: its data density, compared to the original program code, is high enough to display a virus mask (i.e. an area of close neighbourhood in the SOM projection).

Standard anti-virus software can detect variants of a virus only if different virus signatures are available. However, the detection approach in this paper detects viruses independent of any prior knowledge (such as a virus mask). Even polymorphic or encrypted parasitic viruses show a virus mask which can be detected by *VirusDetector*. Since the question of whether or not a program contains virus code is in general unable to answer, *VirusDetector* cannot be perfect. It is therefore a surprisingly good result that in the tests (with 790 files, all infected by a different virus or different version of a virus), *VirusDetector* detected almost 84% of the viruses with a false positive rate of 30%. During the experiments, based on the good detection rate of *VirusDetector*, the confidence in the possibility of detecting unknown viruses increased significantly.

Until now, the classical virus-detection techniques could not deal with unknown viruses (not all, but some). The SOM-based, non-signature-based virus detection complements these standard techniques in that it provides a tool capable of identifying unknown viruses. The combination of signature-based methods with the SOM-based approach can make systems much more secure by making them less vulnerable to infections by unknown viruses.

There is still the macro-virus-detection problem. Although the SOM visualization pattern looks very similar to a virus mask in a NewEXE file, it will always be produced when a document is checked for macro viruses using *VirusDetector*. An approach different from that for parasitic viruses is therefore needed to deal with macro viruses. Even though the macro virus is inserted into a complete document, it is first of all part of a macro and that macro is then saved in the macro area of a document file. Because of this, the inserted macro virus data cannot be differentiated from the macro itself and identified by the SOM neurons. Additionally, the macro virus part is not big enough to produce a significant neighbourhood density and is possibly "hidden behind" low density data. The macro part is simply too small compared to the entire data structure. It will be future work aiming to overcome this problem and be able to deal with macro viruses as well.

Another remaining weakness is the too high false-positive rate of 30%. Even though the experimental results are very promising, additional work must be spent in the future on reducing the relative number of false positives. There is already some scope for that in the approach by adapting the factor value used by *VirusDetector*. This will cause the false-negatives rate to increase, which is partly acceptable. However, it will be necessary to examine other improvements in order to reduce the relative number of false positives.

Nevertheless, if *VirusDetector* can only prevent a single unknown virus from infecting our system (or systems world-wide), the significant research effort spent on developing *VirusDetector* was absolutely worth it.

Chapter 8

Conclusions

The serious problems of life are never fully solved.
If ever they should appear to be so,
it is a sure sign that something has been lost.
The meaning and purpose of a problem seem to lie
not in its solution but in our working at it incessantly.
- C. G. Jung.

In information security, confidentiality, integrity and availability are major pillars to protect and ensure information. Network security problems are the familiar loss of them. Various defence mechanisms need to be held in depth, and risk assessment of data packets must be established in the defence mechanisms. One of the solution technologies is a defence system to protect network-connected resources, which is called a firewall. The more serious network threats, the more important the role of the firewall. Therefore, intelligent firewall technology is indispensable to achieve a high level of protection. This thesis looks for defence mechanisms against network attacks, which use vulnerabilities in network protocols, and risk assessment of data packets, then apply them to network security systems, in particular, the firewall systems. With the concept of an intelligent firewall in mind, potential technologies applicable to an intelligent firewall have been worked and researched. One of the main criticisms of firewalls is that they often create bottlenecks [Ballew(1997)]. To enforce a security policy, firewalls must in some way process all network packets passing through them, and this results in a loss of network performance. This motivates the need for faster firewall technologies, keeping in mind that there are tradeoffs between performance and security, high security requirements must warrant any loss in performance. Therefore, firewalls need to be clever in checking incoming packets efficiently. The goal of this research is to extend the abilities of packet-filtering firewalls aiming to reduce possible problems and attacks by improving firewall technologies.

8.1 Overview

The likelihood of attacks is defined as our confidence to have certain occurrence of specific attacks and attackers' preference to use certain pattern of attacks. The vulnerabilities in network protocols still exist as long as the structure of network and network protocols are not changed completely. Hence, the likelihood of attacks using vulnerabilities in network protocols and mechanisms are still high although new technologies to detect or secure systems' abilities are developed significantly. Therefore, we need intelligent defence mechanisms.

The risk assessment of data packets is defined as that it is about understanding likely threats to network systems and the process of determining whether proposed or existing defence mechanisms are adequate to protect information resources from the threats. The threats to network systems are considered either using vulnerabilities in network protocols and mechanisms or using common network mechanisms with malicious code in data packets. The former can be active attacks including typical denial service attacks, – e.g., ping attacks, SYN flood attacks, land attacks and tear drop attacks – IP spoofing, spams, and denial of service Internet worms. On the other hand, the latter can be passive attacks using social engineering methods including viruses and Internet worms.

There are various existing secure systems; each system has specific defence mechanism against specific attacks. In this thesis, firewall systems are the main concern for risk assessment of data packets. Since firewall systems are not smart enough to protect information resources from the threats, this thesis proposes defence mechanisms to improve firewall technologies. To evaluate packets, filtering, classification, detection, verification, and recognition techniques have been applied to Janus. Based on a packet-filtering firewall, Janus is modelled using adaptive firewall architecture, which can deal with protocol anomaly detection and verification, and email classification. Furthermore, Janus can conduct detecting virus as in attached files without the use of virus signatures.

Fundamental technologies which have been researched in course of this project include software engineering techniques, e.g. the specification description language and abstract state machines, machine learning, especially Naive Bayesian inference, symbolic and algebraic manipulation, and pattern recognition with self-organizing maps.

8.2 Summary

The summary of this thesis is as follows;

- **Protocol Anomaly Detection**: Protocol anomaly refers to all exceptions related to protocol format and protocol behaviour with respect to common practice on the

Internet and standard specifications. However, network attack packets cannot be discovered as being a protocol anomaly, because there exists some odd-looking but legitimate traffic as well. To distinguish protocol anomalies from network traffic, network traffic had been analysed closely using protocol specifications, and then requirements for protocols has been addressed against misuse, based on the protocol specifications. Through that, a packet sanity checker was implemented.

- **TCP Runtime Verification Model**: TCP provides reliable data transfer between different application processes over the network. TCP provides flow control and congestion control as well. Nevertheless, during the past two decades, many security problems of TCP/IP protocol suite have been discovered. Meanwhile, the network hackers created a large number of intrusion methods to exploit those vulnerabilities. Well-known TCP attacks against the current TCP model are SYN flooding and IP spoofing attacks [CERT(2000a)]. These conduct denial-of-service attacks by creating TCP "half open" connections. Any system connected to the Internet and providing TCP-based network services is potentially subject to this attack. Improved OS kernel or stateful inspection firewall/proxy systems support TCP protocol transaction checking against this type of attack. There is one technical solution against SYN flood attack, SYN cookie method, which is generally accepted to this problem with the current IP protocol technology. However, there is no general accepted TCP transaction verification solution, in particular for firewalls, it is necessary to deal with this problem having a practical view on TCP implementation features. Although a standard TCP protocol specification is available, TCP implementations do not fully follow the specification and practical TCP internetworking often violates the protocol standards.

  To identify anomalous transactions in TCP, it has been addressed what current TCP problems are and investigated why practical implementations work differently from the standard TCP specification. Then, to detect anomalous TCP transactions and to check the three-way handshake completed correctly, a TCP verification model has been proposed and was applied to Janus. The TCP verification model does not replace the current TCP specification; it rather provides a verification method of TCP transactions. The TCP verification model is useful to detect anomalous TCP transactions, which can cause system security to be compromised.

- **Email Classification with Naive Bayesian inference and OBDD**: Email is vulnerable to misuse. One such misuse is by email viruses, another is by UBEs also known as Spam. In this project, it is dealt with the propagation of UBE, because the propagation is the potential way for email virus propagation as well. The difference between UBE and Email viruses is whether or not malicious content s present. To detect misused emails, and to estimate the probability of whether the mail is abnormal, the structure of emails has been examined and the header field, the sender field, and the recipient field of each email transferred using the

SMTP protocol, have been checked. Then a Naive Bayesian classifier has been built representing the probability of an email attachment of being malicious. Prior probabilities to malicious mail have been assigned and the likelihood of being malicious of an executable attachment has been estimated. Then an exact algorithm for Bayesian inference is used, and the Naive Bayesian classifier is converted into an OBDD. Since Janus should deal with packets fast enough, the OBDD has been designed with predefined properties. The Bayesian-OBDD based model is capable of drawing conclusions about the potential maliciousness of incoming email packets. Based on the estimation of a packet's potential maliciousness and using a particular security policy of the protected network, a data packet can then either be dropped or it passes.

- **Virus Detection and Recognition with Self-Organizing Maps**: To recognize virus patterns in virus-infected executable files, without virus signatures, a self-organizing map (SOM) is used to visualize data densities through the SOM projection. Without using virus signatures, this SOM projection tells us the structure of a virus-infected executable file. Besides we can detect virus-infected executable files independent of whether the virus is known or not. A virus is a part of the file, which was inserted in a certain way, and this virus code shows a different structure when compared to the original program code it infected. Thus, virus code is distinguishable from the remaining program code, whether it is encrypted or not. Therefore, the SOM produces a particular pattern, so-called the *virus mask* when a file is virus-infected. The virus mask can prove decisive in establishing the existence of virus in virus-infected files. Anti-virus software can detect variants of a virus only by different virus signatures. This research can be applied to anti-virus detection without any knowledge of virus signatures. It is therefore applicable to the detection of unknown new viruses.

- **Packet Filtering and Classification**: Packet classification performs a matching between IP packets and filters in order to find which filter matches the packet. It should be fast enough to allow the classification algorithm being executed several times and still forward packets at line speed. Packet filtering is a special case of packet classification. Although Janus performs packet classification, the purpose of this project is improving firewall technologies rather than creating a new classification algorithm to improve the filtering speed. Therefore, a quite simple packet-filtering mechanism has been implemented using double-linked lists and the Booyer-Moore pattern matching algorithm.

- **Janus Firewall**: Janus is an adaptive firewall model. After developing technologies applicable to the recognition of anomalies in data packets, those have been implemented in Janus. Janus does not require the presence of the TCP/IP stack. To reduce the scope for security holes in protocol stacks, it has been necessary to recompile the Linux kernel with support for network cards, but without any

TCP/IP networking support. Janus is designed to protect of any server systems against any internal or external network attacks. The current version of Janus is placed inside the university's firewalls with a separated subnetwork routing setup on an Internet connection. This causes Janus to act as a firewall/router on the subnetwork since all traffic travels through it. The current IP settings and working environment of the Janus box are that of a single server environment.

- **Janus VirusDetector**: A non-signature-based virus detection program, *VirusDetector* is presented. Unlike classical virus detection techniques using virus signatures, this SOM-based approach can detect virus-infected files without any prior knowledge of virus signatures. Exploiting the fact that virus code is inserted into a complete file which was built using a certain compiler, an untrained SOM can be trained in one go with a single virus-infected file and will then present an area of high density data, identifying the virus code through SOM projection. *VirusDetector* has been tested on 790 different virus-infected files, including polymorphic and encrypted viruses. It detects viruses without any prior knowledge — e.g. without knowledge of virus signatures or similar features — and is therefore assumed to be highly applicable to the detection of new, unknown viruses. This non-signature-based virus detection approach was capable of detecting 84% of the virus-infected files in the sample set, which included polymorphic and encrypted viruses. The false positive rate was 30%. The combination of the classical virus detection technique for known viruses and this SOM-based technique for unknown viruses can help systems even more secure.

8.3 Future Work

- **Analyzing Firewall Rules**: For firewalls, they must also be configured properly. Firewall configurations are often written in a low-level language, which is very hard to understand. For instance, the order of the rules is often very important. Thus, it is often quite difficult to find out which connections and services are actually allowed by the configuration.

  This brings up two related problems: how to express the organization's security policy in a language understood by the firewall and finding out what a given firewall configuration actually does. This second problem often occurs when a new network administrator takes over, for instance, or when a third party is performing a technical security audit for the organization.

- **Encryption**: Suitable encryption can defend against some attacks. However, encryption devices are expensive, often slow, hard to administer, and uncommon in the civilian sector. There are different ways to apply encryption; each has its strengths and weaknesses. A comprehensive treatment of encryption is beyond

the scope of this thesis. Nevertheless, encrypted messages can contain malicious information that cannot be detected by the current intelligent firewall. Message encryption is a problem, especially for network-based intrusion systems. Encryption makes the practice of looking for particular patterns in packet bodies futile. Useful analysis can be performed only after the message has been decrypted on the target host, and this often occurs within a specific application. Driven by commercial and defence needs, greater emphasis needs to be placed on host-based or application-based intrusion systems that have the ability to view message content even if the message is encrypted in transit. However, IPSec provides a mechanism, encapsulating security payload, which can be used to hide both the contents and addresses of network packets between firewalls. This renders the actual source, destination, and contents of the packet opaque while they are in transit between firewalls.

- **Network Planning/Configuration/Optimisation**: Janus does not have logical configuration tools to make sure the configuration does not contain any logical errors. The logic should cover the current network environment and its configuration. Therefore, the tool will have to support a planning or optimising network configuration avoiding logical flaws.

- **Denial of Service attack**: To deal with denial of service attacks, network anomaly detection and traffic classification are future add-ons to Janus. This requires research such as the comparison of attack traffic with legitimate traffic, classifying network-based application response, and identifying network traffic patterns.

- **Internet Worms**: Email classification of Janus cannot fully protect against Internet Worms, if a worm looks like legitimate traffic. To overcome this, Janus will have to cooperate with network Intrusion detection systems. In addition, to protect systems from Internet worms, firewall access and filtering, attack identification and analysis, and identification of network traffic patterns are required.

- **Janus VirusDetector**: SOM's ability to gather likelihood data together gave us benefit to visualize and to detect virus-infected files. Since virus codes could not hide their own features through SOM projection, it was great advantage how virus codes affected the whole file projection, and through this projection, we could see virus masks without knowing any virus-signature knowledge. Then VirusDetector implementation presented how to detect the virus masks, and what kind of process was taken to build VirusDetector. In addition, several results of visualization and detection, and the evaluation of VirusDetector were presented. Without virus-signature knowledge, about 84 % of detection ratio with 30 % of false positive is encouraging. Still much of experiments are necessary to improve VirusDetector, and macro virus detection part would be a future work.

'It's a dangerous business, Frodo, going out of your door,'
he used to say.
'You step into the Road, and if you don't keep your feet,
there is no knowing where you might be swept off to.'
- The Lord of The Rings, J.R.R. Tolkien

Appendix A

TCP Runtime Verification Model SDL Specification

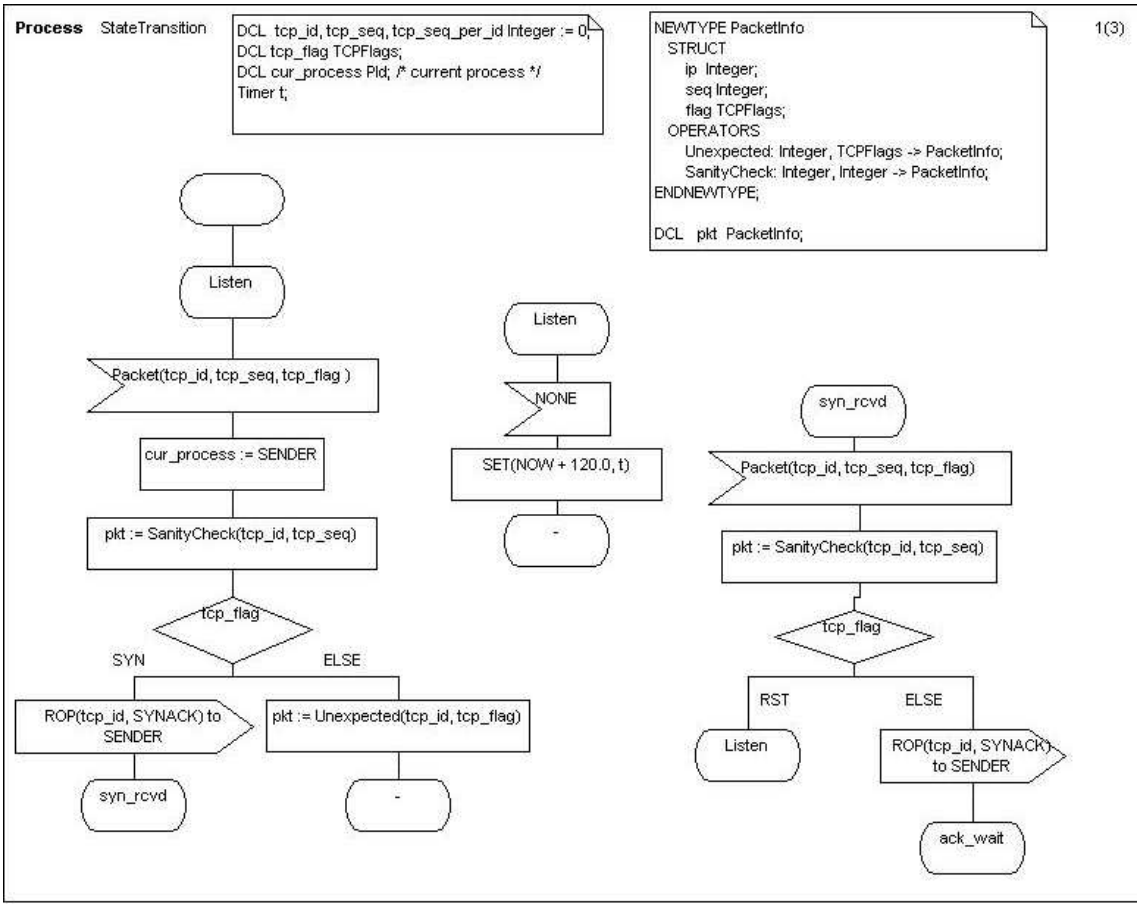

FIGURE A.1: Process StateTransition of the TCP Protocol State Machine in SDL

SYSTEM StateMachine ;
NEWTYPE TCPFlags
LITERALS SYN, ACK, FIN, RST,

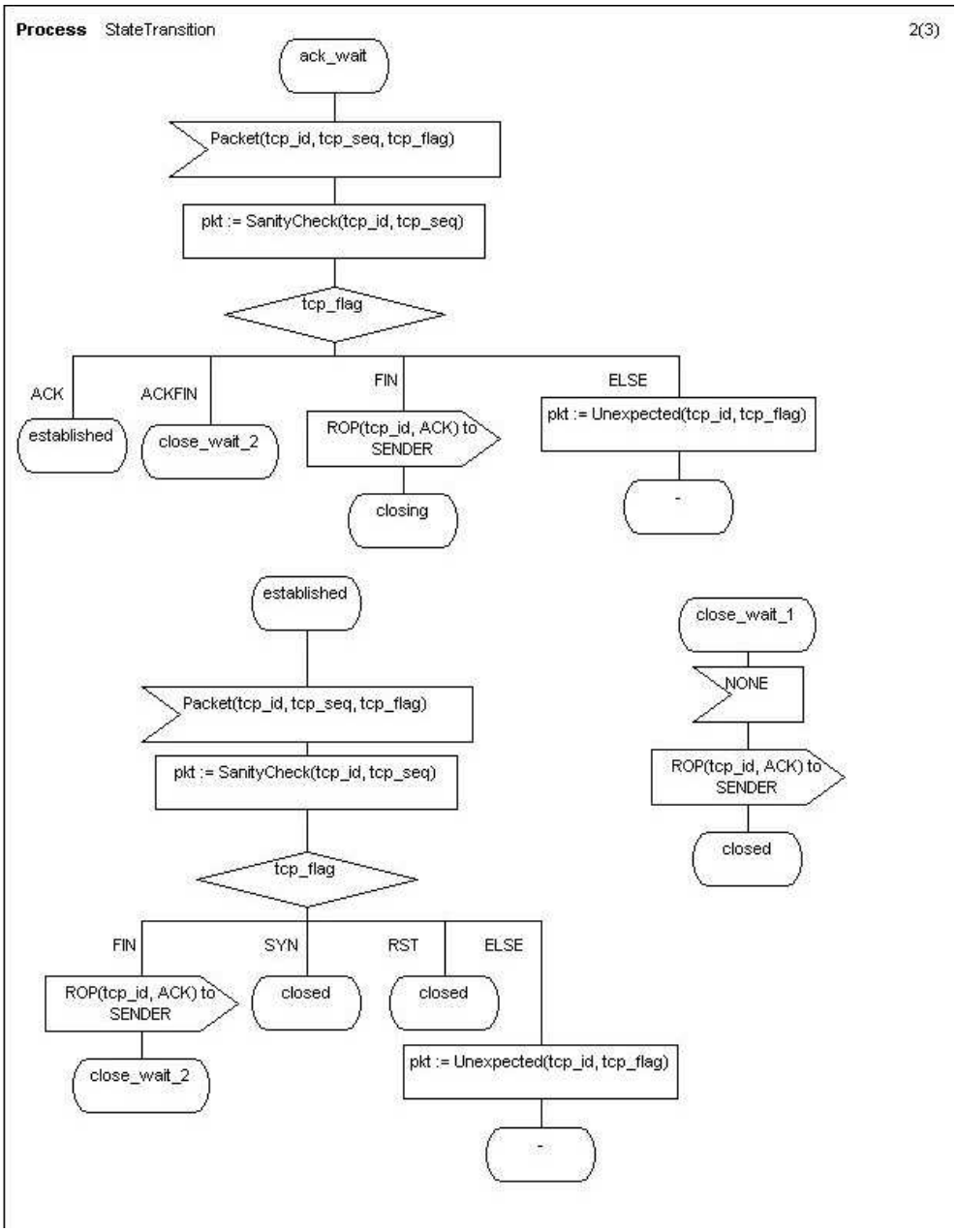

FIGURE A.2: Process StateTransition of the TCP Protocol State Machine in SDL

```
SYNACK, /* SYNACK = SYN + ACK */
ACKFIN, /* ACKFIN = ACK + FIN*/
FFF;
ENDNEWTYPE;

SIGNAL Packet(Integer, Integer, TCPFlags); /*tcp_id, tcp_seq, flags*/
SIGNAL ROP(Integer, TCPFlags); /* Result of Packet, tcp_id, flags */


CHANNEL channelout
NODELAY FROM tcpstate TO ENV WITH ROP ;
ENDCHANNEL;
```

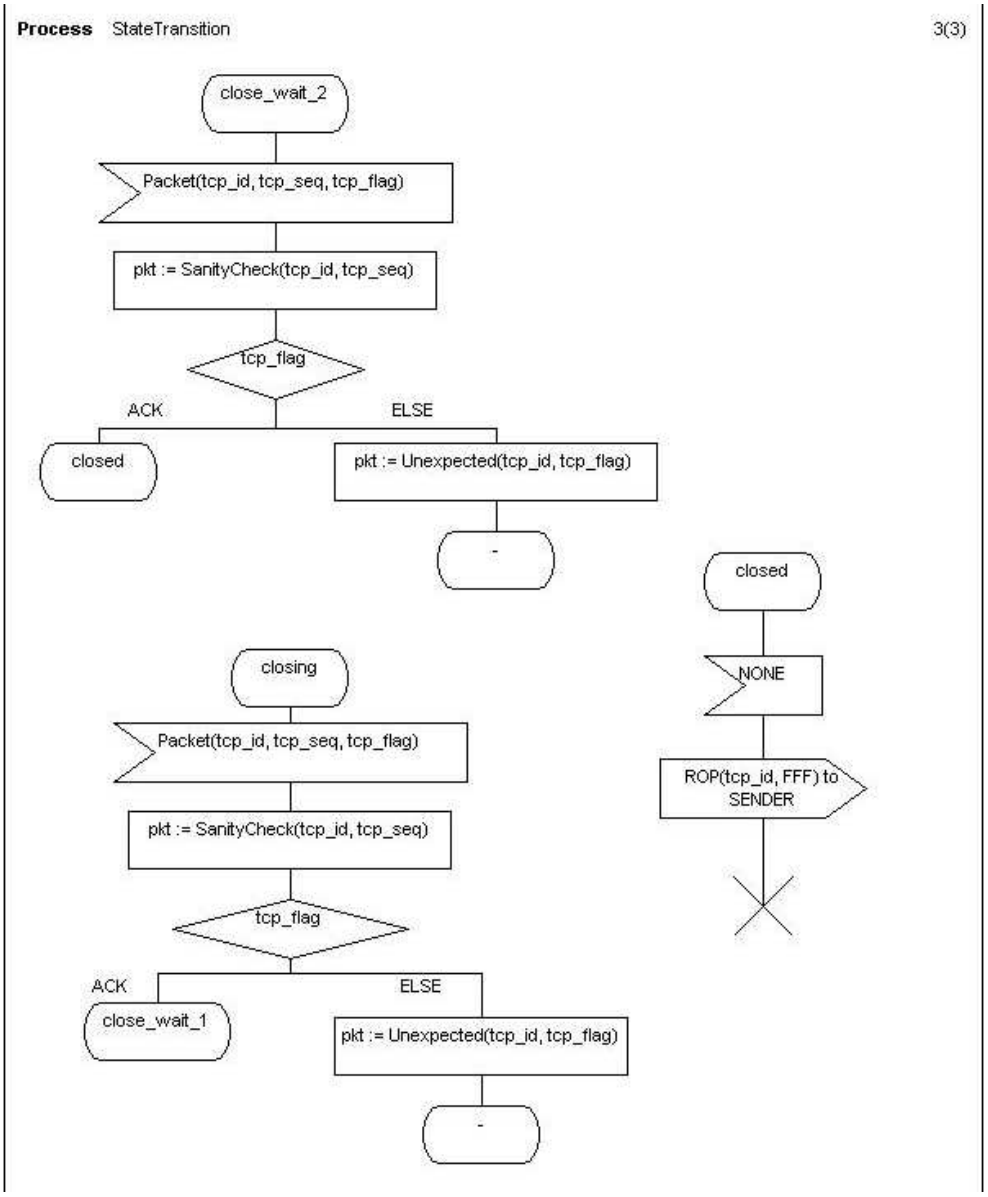

FIGURE A.3: Process StateTransition of the TCP Protocol State Machine in SDL

CHANNEL channelin
NODELAY FROM ENV TO tcpstate WITH Packet ;
ENDCHANNEL;

BLOCK tcpstate REFERENCED;
ENDSYSTEM;

PROCESS<<SYSTEM StateMachine/BLOCK TCPState>> StateTransition ;
NEWTYPE PacketInfo
STRUCT
ip Integer;
seq Integer;
flag TCPFlags;
OPERATORS

```
Unexpected: Integer, TCPFlags -¿ PacketInfo;
SanityCheck: Integer, Integer -¿ PacketInfo;
ENDNEWTYPE;


DCL pkt PacketInfo;
DCL tcp_id, tcp_seq, tcp_seq_per_id Integer := 0;
DCL tcp_flag TCPFlags;
DCL cur_process PId; /* current process */
Timer t;

START;
NEXTSTATE Listen ;
STATE syn_rcvd ;
INPUT Packet(tcp_id, tcp_seq, tcp_flag) ;
TASK pkt := SanityCheck(tcp_id, tcp_seq) ;
DECISION tcp_flag ;
( RST ):
NEXTSTATE Listen ;
ELSE:
OUTPUT ROP(tcp_id, SYNACK) to SENDER ;
NEXTSTATE ack_wait ;
ENDDECISION;
ENDSTATE;

STATE Listen ;
INPUT Packet(tcp_id, tcp_seq, tcp_flag ) ;
TASK cur_process := SENDER ;
TASK pkt := SanityCheck(tcp_id, tcp_seq) ;
DECISION tcp_flag ;
( SYN ):
OUTPUT ROP(tcp_id, SYNACK) to SENDER ;
NEXTSTATE syn_rcvd ;
ELSE:
TASK pkt := Unexpected(tcp_id, tcp_flag) ;
NEXTSTATE - ;
ENDDECISION;
ENDSTATE;

STATE Listen ;
INPUT NONE;
SET(NOW + 120.0, t) ;
NEXTSTATE - ;
ENDSTATE;

STATE close_wait_1 ;
INPUT NONE;
OUTPUT ROP(tcp_id, ACK) to SENDER ;
NEXTSTATE closed ;
```

```
ENDSTATE;

STATE established ;
INPUT Packet(tcp_id, tcp_seq, tcp_flag) ;
TASK pkt := SanityCheck(tcp_id, tcp_seq) ;
DECISION tcp_flag ;
( RST ):
NEXTSTATE closed ;
( SYN ):
NEXTSTATE closed ;
( FIN ):
OUTPUT ROP(tcp_id, ACK) to SENDER ;
NEXTSTATE close_wait_2 ;
ELSE:
TASK pkt := Unexpected(tcp_id, tcp_flag) ;
NEXTSTATE - ;
ENDDECISION;
ENDSTATE;

STATE ack_wait ;
INPUT Packet(tcp_id, tcp_seq, tcp_flag) ;
TASK pkt := SanityCheck(tcp_id, tcp_seq) ;
DECISION tcp_flag ;
( FIN ):
OUTPUT ROP(tcp_id, ACK) to SENDER ;
NEXTSTATE closing ;
( ACKFIN ):
NEXTSTATE close_wait_2 ;
( ACK ):
NEXTSTATE established ;
ELSE:
TASK pkt := Unexpected(tcp_id, tcp_flag) ;
NEXTSTATE - ;
ENDDECISION;
ENDSTATE;

STATE close_wait_2 ;
INPUT Packet(tcp_id, tcp_seq, tcp_flag) ;
TASK pkt := SanityCheck(tcp_id, tcp_seq) ;
DECISION tcp_flag ;
( ACK ):
NEXTSTATE closed ;
ELSE:
TASK pkt := Unexpected(tcp_id, tcp_flag) ;
NEXTSTATE - ;
ENDDECISION;
ENDSTATE;
```

```
STATE closed ;
INPUT NONE;
OUTPUT ROP(tcp_id, FFF) to SENDER ;
STOP;
ENDSTATE;

STATE closing ;
INPUT Packet(tcp_id, tcp_seq, tcp_flag) ;
TASK pkt := SanityCheck(tcp_id, tcp_seq) ;
DECISION tcp_flag ;
( ACK ):
NEXTSTATE close_wait_1 ;
ELSE:
TASK pkt := Unexpected(tcp_id, tcp_flag) ;
NEXTSTATE - ;
ENDDECISION;
ENDSTATE;
ENDPROCESS;

BLOCK<<SYSTEM StateMachine>> TCPState ;
SIGNALROUTE sr2
FROM statetransition TO ENV WITH ROP ;
SIGNALROUTE sr1
FROM ENV TO statetransition WITH Packet ;
PROCESS statetransition REFERENCED;
CONNECT ChannelIn AND sr1;
CONNECT ChannelOut AND sr2;
ENDBLOCK;
```

Appendix B

Background of Packet Classification and Filtering

B.1 Packet Filtering

Packet filtering is a special case of packet classification. Packet classification determines what class a packet belongs to, based on the fields in its header and a set of classification rules. The class of the packet determines what action should be performed on it. The packet filter checks each incoming and outgoing packet, and makes sure that the destinations and sources of the packets are trustworthy. Any packet from a source or to a destination that is not trustworthy, is prevented from passing through the network. Packet filters work by looking up a rulebase which lists sources and destinations that should be allowed, and those that should be blocked. Packet filters uses a rulebase to determine whether to forward or discard a packet. However, rule lookup time can increase network latency significantly [Cheswick and Bellovin(1994)]. Rule lookup latency can especially be a problem if the rulebase is large, since the resulting latency can reduce response time unacceptably. Furthermore, in high-speed networks rule lookup can be a bottleneck and reduce network throughput substantially [Ellermann and Benecke(1998)].

Although the ability to filter and detect packet is worthy for firewalls, performance cannot be ignored. Thus, both hardware and software-based packet filter researches exist. Software-based packet filters are especially prone to the latency problem [Newman(1999)]. There are three reasons for this. Firstly, some filters simply search the rulebase sequentially from beginning to end; this is very inefficient as it is possible to optimise the search using specialised data structures and algorithms. Secondly, the filtering code runs on a general-purpose CPU. Hence, no matter how effectively the data structures and algorithms are optimised, there is a limit to the performance that can be achieved. Thirdly, software solutions suffer from a high overhead. For example, several layers of hardware and software have to be traversed during rule lookup: the packet must be transferred

from the physical cable to the network card, through the protocol stack and to the filtering software, and back to the network again, traversing many buffers and buses in the process. Other overheads result from operating system tasks such as interrupt handling.

Hardware-based packet filters using application specific integrated circuits (ASICs) [1] can decrease lookup times significantly, compared to software-based filters. The lookup algorithms can be implemented on the ASICs making them extremely fast. Also, most of the overhead that software suffers is eliminated, because the hardware can be integrated directly into the network, resulting in fewer layers to traverse during lookups. This is the case with Fore Systems' ASIC-based gigabit Ethernet switches, which are reportedly 1000 times faster than software filters [Newman(1999)]. However, there are two problems with hardware-based filters. Firstly, they are expensive, making the resulting solution viable only for large corporations. A second disadvantage of filter ASICs is that the algorithms hardwired into the ASICs cannot be modified once manufactured. Changing an ASIC design is time consuming and has high non-recurring engineering (NRE) costs [Salcic and Smailagic(1997)]. However, there is a possible solution for the problem of rule lookup latency using field programmable gate arrays (FPGAs). FPGAs are a form of programmable logic that can be used to create custom hardware. Using FPGAs, it should be possible to develop a high-speed, hardware-based packet filter. FPGAs are also a kind of ASIC, but are becoming much cheaper than other ASICs and are replacing them in many areas. For example, a major FPGA manufacturer, Xilinx Inc., has introduced the Spartan family of FPGAs specifically to replace other ASICs [Xilinux(2001)]. FPGAs can also be cheaply and quickly reprogrammed/reconfigured, making it easy to change an existing solution [Salcic and Smailagic(1997)].

Although software implementation are unlikely to ever match the speed of hardware implementations, many packet filters in use are software-based, and this justifies the efforts made in software-based improvements. Software-based efforts at improving packet filtering include designing various alternative internal representations of access lists that facilitate faster lookup. This research adopts a software-based approach.

B.1.1 Packet filter rules

The security policy for a packet filter is specified by a set of rules. Each rule has the structure: if condition then action, where the condition defines a logical statement based on fields within the packet header, and the action specifies whether a packet matching that rule should be accepted or rejected. A set of rules is known as an access control list, or simply, access list [Ballew(1997)]. Semantically, the rules of an access list are considered sequentially, so the condition of the first rule that matches a packet will

[1] Two ICs that might or might not be considered ASICs are a controller chip for a PC and a chip for a modem. Both of these examples are specific to an application but are sold to many different system vendors. ASICs such as these are sometimes called application-specific standard products (ASSPs).

determine the action to be taken.

This format for representing security policies has both advantages and disadvantages. The simplicity of its structure makes individual rules easy to specify. By specifying source or destination addresses as well as ports in the condition of a rule, it is fairly easy to define security policies that allow access for specific services on specific hosts [Cheswick and Bellovin(1994)]. However, the semantics of access lists results in the order in which rules appear in the list being extremely significant, the interaction between the rules in an access list can make the interpretation of what types of packets it accepts difficult [Oppliger(1998)].

For this reason, packet-filtering implementations traditionally represent access lists internally as a linear list of rules that is a direct translation of the original access list. The decision-making process that determines whether a particular packet passing through the packet filter should be accepted or rejected is called lookup and is closely tied to the semantics of access lists. During lookup, the rules of the access list are applied sequentially to the packet until a matching rule is found, at which point the corresponding action is taken. If no rule matches, the default rule is applied, which is usually to reject all packets. In general, no context is kept, so this lookup process must be repeated for every packet [Cheswick and Bellovin(1994)].

Packet filter rules consist of two parts: an action and an associated condition [Oppliger(1998)]. The action specifies whether to deny or permit the packet. The condition specifies the selection criteria that the packet should meet in order for the action to be taken on it. The selection criteria could include, the source of the packet, its destination and/or its protocol type.

The rule description language is specific to a particular system. For example, a typical CISCO access rule might be like Table B.1:

TABLE B.1: An general form and an example of a CISCO access rule.

list#	action	protocol	src-addr	src-mask	dest-addr	dest-mask	port-range
101	permit	tcp	20.9.17.8	0.0.0.0	121.11.27.20	0.0.0.0	range 21 25

Where,

- list-number is the number of the access list to which the rule belongs. There could be several access lists on the system.
- action specifies whether to permit or deny the packet.

- src-addr and dest-addr are four segment dotted-decimal IP addresses of the source and destination of the packet, respectively.

- src-mask and dest-mask act like wild cards and specify which bits of the source and destination addresses to ignore and which to match. They allow the administrator to specify several possible matching addresses. For example, a source address of 134.21.54.111 with mask 0.0.0.0, indicates that the packet's source address should match exactly. On the other hand, a mask of 0.0.255.255 indicates that any packet's source address that has the prefix 134.21, for example, 134.21.53.114 will match.

- port-range gives the range of port addresses allowed.

The list of packet filter rules are searched sequentially to determine one that applies to a given packet. Hence, the ordering of the rules matters.

B.1.2 Theoretical Bounds on Packet Classification

In general, packet classification algorithms trade off space for time, or vice versa. Traditional packet filters use a sequential algorithm to classify packets. The algorithm has linear time complexity in the number of rules and constant space requirements [2]. This is considered very efficient in terms of space, but not extremely efficient in terms of time. On the other hand, by pre-computing the matching rule for all 2^S possible inputs, where S is the number of bits of interest in the packet header, in a table, the lookup time is constant. However, the memory requirements grow exponentially with S resulting in unreasonable memory requirement even when S is relatively small. Thus, the real challenge lies in finding a solution that is efficient in terms of both space and time.

B.2 Related Work on Packet Classification & Filtering

Packet classification is the problem of finding the least cost rule that matches a packet. In the case of packet filtering as a means of access control, the cost of a rule can be thought of as its position in the access list. This section presents software- and hardware-based classification and internal rule representations proposed by previous research work in these areas. Especially, software-based methods address table-driven methods and specialised data structure for packet classification and filtering.

[2]Space complexity measures the amount of extra storage required by the algorithm, not including the storage required for the representation of the input, which in this case is the access list.

B.2.1 Table-driven Methods

Table-driven methods are characterized by algorithms that preprocess the filters into some tabular representation. In some cases, hashing is used to improve performance. Table-driven methods tend to be sensitive to the type of data that they receive; they rely on the structure and redundancy found in typical access lists, so that access lists with similar looking rules result in better performance than access lists with arbitrary rules. Data that is not well behaved may cause either poor memory or time performance.

B.2.1.1 Tuple Space Search

Tuple Space Search (TSS) is a general packet classification, however, the research focused on 5-dimensional filters for the experiments; IP source, IP destination, protocol type, source port number, and destination port number. TSS defines a tuple T as a vector of K lengths; K is the number of fields for filtering. For example, [8, 16, 8, 0, 16] is a 5-dimensional tuple, whose IP source field is an 8-bit prefix, IP destination field is a 16-bit prefix, and so on. A filter F belongs or maps to tuple T if the ith field of F is specified to exactly T[i] bits. For example, 2-dimensional filters $F_1 = (01*, 111*)$ and $F_2 = (11*, 010*)$ both map to the tuple [2, 3].

Tuples require all fields of a filter to be specified as a length. While IP addresses are always specified using prefixes, port numbers are not. Port numbers are usually specified using ranges, e.g., [0, 1024]. TSS gets around this requirement by using nesting level and RangeId each to simulate prefix length and prefix of IP addresses. For example, we have three ranges; $F_1 = (0, 65535), F_2 = (0, 1023), F_3 = (1024, 65535)$. F_1 has nesting level of 0 and RangeId of 0. F_2 and F_3 are nested from F_1, thus their nesting level is 1, and they receive RangeId of 0 and 1, respectively.

With the get around explained above, each filter can now be mapped to a particular tuple T in a hash table $Hashtable(T)$ with the concatenated prefix-es and RangeId-es as its hash key. Probing a tuple T involves concatenating the required number of bits from the packet P as specified by T and then doing a hash in $Hashtable(T)$. Searching for a matched filter for a given packet P is performed by linearly probes all the tuple in the tuple set. If more than one matching filters were found, TSS picks the least cost filter. The search cost is proportional to m, the number of distinct tuples, which can be up to N, the number of filters in database. However, the previous observation showed that N tends to be much larger than m. Update cost (inserting and deleting a filter) for tuple space search is also small, only one hash access. Thus, we can say that tuple space search performs much better than linear search.

Srinivasan et al [V. Srinivasan and Varghese(1999)] shows several improvements for the basic tuple space search ; Tuple Pruning and Rectangle Search. Tuple Pruning is motivated by the observation that in real filter databases there is no address D has more

than 6 matching prefixes. If a 2-dimensional filter is formed from that database, then if we first find the longest destination match and the longest source match, there are only at most 6 x 6 = 36 possible tuples that are compatible with the individual destination and source matches. This is very small compared to the maximum number of possible tuples for IP source-destination pair, which is 36 x 36 = 1024. For instance, consider a 2-dimensional filter database for source S and destination D addresses. Suppose $D = 1010*$, and all the filters whose destination is a prefix of D belong to tuples [1, 4], [1, 1], and [2, 3]. Then, the tuple list of D contains these 3 tuples. Similarly, suppose $S = 0010*$, and all filters whose source is a prefix of S belong to tuples [2, 4], [1, 1], and [2, 5]. Searching for the matching filter for a packet P computes the longest matching prefix PD and PS for destination and source addresses of P. The next step is to take the tuple lists stored with PD and PS, find their common intersection, and probe into that intersection. If $PD = D$ and $PS = S$ as above, the intersection list only includes [1, 1], thus we only probe into one tuple.

Rectangle Search is an improvement from the basic tuple space search for 2-D filter database. Rectangle Search works to cut the search time by using precomputation and markers. Search time is now down from W^2 hash accesses of basic tuple space search into $2W$ accesses, W is the bit width of address. Srinvasan et al also shows that this algorithm is optimal for 2-D filter database. When a filter is added to the database, it leaves a marker at all the tuples to its left in its row. So a filter in the tuple $[i, j]$ leaves a marker in tuples $[i, j-1], [i, j-3], ...[i, 1]$. Each filter or marker also precomputes the least cost filter matching it from among the tuples above it in its column. That is, a filter or marker in tuple $[i, j]$ precomputes the least cost filter matching it from the tuples $[i-1, j], [i-2, j], ..., [1, j]$. This is the marking and precomputation strategy for rectangle search.

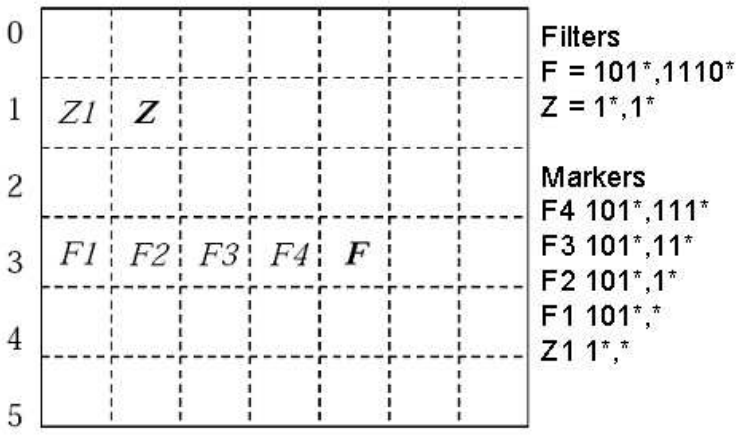

FIGURE B.1: Illustration of markers and precomputation

Figure B.1 shows an example of precomputation and markers, using two filters F and Z. The marker F_2 precomputes the best matching filter among the entries in the column above it, which in this example is Z. The search strategy for this algorithm starts by

probing the lower-left tuple namely, $[W, 1]$. At each tuple, if the probe returns a match, the search moves to the next tuple in the right. If there is no match, the search moves up one row in the same column (see Figure B.2.). When a match is found, it is an indication that there is a filter on the right of the current tuple, thus it is not necessary to probe into the tuples above the current one. However, in case of no match, then there is no filter in the tuples on the right, therefore the search can eliminate the tuples on that row. The search terminates when we reach the rightmost column or the first row.

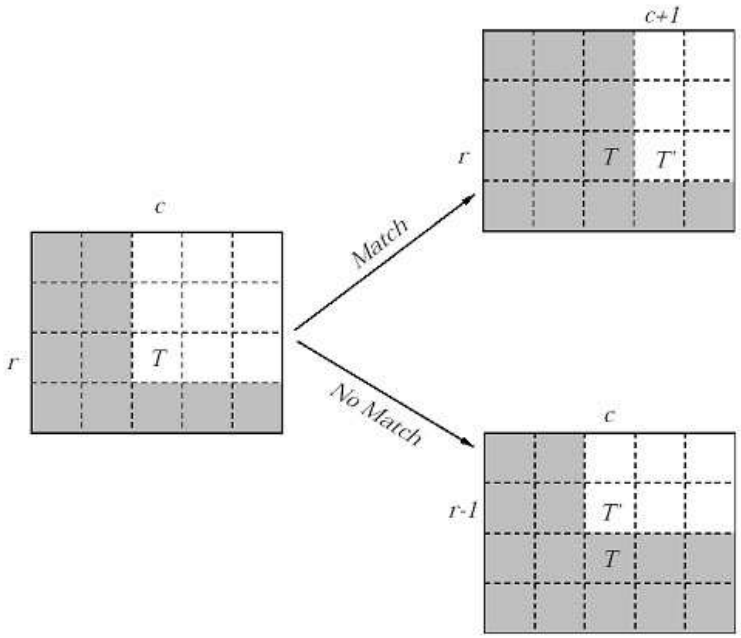

FIGURE B.2: Illustration of search strategy

B.2.1.2 Multi-dimensional Range Matching

Multi-dimensional range matching [Lakshman and Stiliadis(1998)] is a technique that solves the classification problem as the intersection of a number of simpler problems. Given n as the number of rules and k as the dimensionality of the filter, the lookup algorithm is actually linear in n. However, it makes use of bit-parallelism - the time taken for a single bit operation is the same as the time taken for a word operation, so multiple bit operations can be executed in parallel at no extra cost - to reduce the actual lookup time.

The classification algorithm is a geometric algorithm that views the rules encompassing a rectangular area in k dimensions, the packets as points in the k-dimensional space, and the lookup algorithm as finding the least cost rectangle that contains the point since the rectangles may overlap. The preprocessing part of the algorithm projects the edges of the rectangles to their corresponding axes, cutting each axis into a number of intervals. In the worst case, there may be $2n + 1$ intervals in each dimension. For each interval

in each dimension, a bitmap is created that has its ith bit set if and only if rectangle i overlaps with that interval.

The lookup algorithm then works as follows. When a packet arrives, the intervals that contain this point are located for each axis. Then the bitmaps for those intervals are ANDed bitwise, and the first bit that is set in the resulting bitmap corresponds to the least cost rectangle. This is due to the fact that the rectangles are numbered based on their priorities, in the order that the rules appear in the access list. While this algorithm is an improvement on the traditional linear algorithm, its strength lies in hardware implementations that can perform much more of the processing in parallel.

B.2.1.3 Scalable High Speed IP Routing Lookup

Waldvogel et al [Marcel Waldvogel and Plattner(1997)] describes a new algorithm for best matching prefix using binary search on hash tables organized by prefix lengths. There are three significant ideas of this algorithm; using hashing to check whether an address D matches any prefix of a particular length, binary search to reduce the number of searches from linear to logarithmic, and precomputation to prevent backtracking in case of failures in the binary search of range.

Hashing idea is to look for all prefixes of a certain length L using hashing and use multiple hashes to find the best matching prefix, starting with the largest value of L and working backward. For example, consider a routing table of four prefix entries, each with prefix length of 4, 8, 8 and 10. Each of the entries would be stored in a hash table that corresponds to its prefix length (Figure B.3). The hash tables are stored as a sorted array, so for this example, the array has three entries.

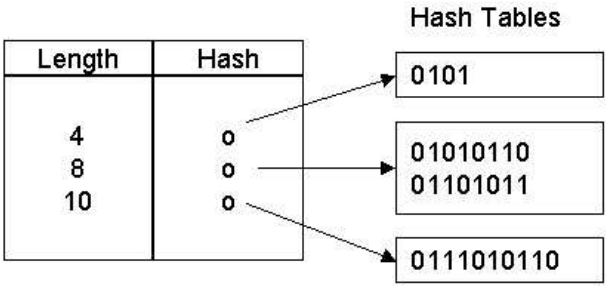

FIGURE B.3: Hash tables for prefix lengths

Searching for address D, we walk through each hash table in that array starting from the largest value l, i.e., 10 on the example, extracting the first l bits of D to get its prefix of D. We then search the hash table using that prefix as the key. If we find a prefix, then we have found the best matching prefix (BMP) and the search terminates; otherwise, if we find nothing, we move to the next entry of the array.

To illustrate the binary search strategy, suppose we have three prefixes P1 = 0, P2 =

00, P3 = 111. Storing the prefixes in hash tables and sorting the array, we have Figure B.4(b). If we search for 111, binary search (a) would start at the middle of the hash table and search for 11 in the hash table containing P2. This search would fail and have no pointer that it should search in the longer prefix tables to find the BMP. To correct the search, we need to put the marker for prefix P3 in this table, thus the lookup for 11 would succeed and binary search would know that it should search for a match in the longer prefix table.

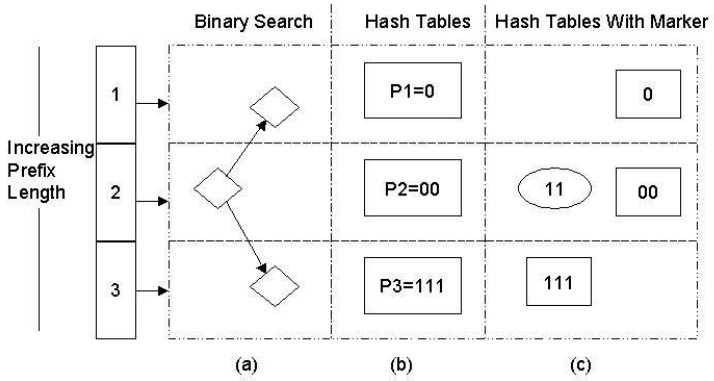

FIGURE B.4: Binary search on hash tables

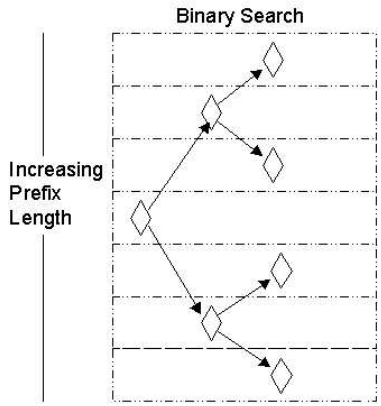

FIGURE B.5: Binary search on trie levels

The hash tables containing prefixes and markers can be thought as a trie where each hash table is a level of a trie that corresponds to nodes of a certain prefix length (see Figure B.5). Binary search in this trie starts on the median level of the trie and depending on the result of hash lookup on that level, the search will continue to the level of shorter or longer prefix length.

A naive implementation of this algorithm will take linear time. While Markers can point to the BMP, they can also cause the search to follow false leads, which may fail. When this happens, we would have to modify the binary search to backtrack and search the upper half of the level of failure, and that would lead to linear time. To avoid the backtracking problem, we need to use precomputation when inserting markers. Suppose we insert a marker M to the hash table, M would have to record the best matching

prefix of the marker (M.bmp). With this variable, now the binary search remembers the value of M.bmp whenever a lookup produces a match. If the search in the lower half produces a failure, search procedure does not need to backtrack, since it remembers the best matching prefix from M.bmp.

B.2.2 Specialised Data Structures

Classification algorithms that use specialised data structures to represent the access lists internally tend to be less sensitive to the types of lists they encounter compared to the table-driven methods. Data structures used to represent access lists are often graph-based and tend to perform quite well in practice.

B.2.2.1 Grid of Tries

A trie is a binary branching tree whose edges are labelled either 0 or 1, and so the path from the root to a particular node corresponds to the unique bit sequence of the node [Cheung and McCanne(1999)]. Tries are associated with routing algorithms that match routing table rules according to the destination address of the packet. This structure has been extended to support matching on two fields instead of just one with a grid of tries [Venkatachary Srinivasan and Waldvogel(1998)]. Essentially, after the trie for the first field has been constructed, tries for the second field are constructed off relevant nodes in the original trie. This final data structure behaves somewhat like an automaton scanning one bit of the input at a time and never backtracking. Unfortunately, this scheme does not extend to multi-dimensional filters, since it implicitly builds longest prefix matching into the structure and has the constraint that the specified bits in each prefix must be contiguous and start matching from the start of the input string. So while this structure is useful for applications such as multicast forwarding, it is not very useful for general packet filtering applications.

B.2.2.2 Expression Trees

The earliest work on packet classification demultiplexing used expression trees to represent the rules [J. Mogul and Accetta(1987)]. This data structure was a natural choice due to the stack-based implementation used. An expression tree is a binary tree that has Boolean predicates at its leaf nodes and Boolean operations, e.g., AND and OR, at its internal nodes. The value of the expression is given by performing an in-order traversal of the tree. One of the major problems with this approach is that it does not maintain state, meaning that the packet may need to be parsed several times to evaluate the expression, resulting in redundant computations. The original implementation using

this data structure is called CSPF (CMU / Stanford Packet Filter) and has lookup times that grow linearly with the number of rules.

B.2.2.3 Binary Directed Acyclic Graphs

CSPF is the Berkeley packet filter (BPF) [McCanne and Jacobson(1993)] that improves on the original design by using control flow graphs (CFGs) to represent the rules instead of expression trees. CFGs are directed, acyclic graphs whose nodes represent Boolean predicates such as destination = foo, and edges represent control transfers (one edge is traversed if the corresponding predicate is true; the other edge is traversed if it is false). Unlike expression trees, the CFG model allows state information to be implicitly built into the data structure, which avoids recomputing identical predicates. Various optimisations have been applied to the original CFGs that further attempt to reduce redundant computations which has the effect of reducing both the lookup times and memory requirements [M. L. Bailey and Sarkar(1994)] [A. Begel and Graham(1999)] [M. Yuhara and Moss(1994)]. Although these packet filters perform much better than the original CSPF, no theoretical results have been published to describe time or memory requirements in general.

Baboescu and Varghese [Baboescu and Varghese(2001)] describe a scheme called Aggregate Bit Vector (ABV). The aim of the scheme is to provide scalable packet classification, e.g., 100,000 rules, to handle large filters while also providing efficient classification times on generic CPUs. The scheme is an extension of the bit vector search algorithm (BV) described in [Lakshman and Stiliadis(1998)]. The first optimisation of the BV scheme consists of minimizing the number of unused bits in the bit vectors, by taking advantage of the observation that the number of rules overlapping in a filter is likely to be small. This is technique referred to as aggregation. Secondly, to take full advantage of using aggregation the order of the rules is rearranged. However, again due to the issues of overlapping rules, it is not possible. Modifying the BV scheme to first find all matches and then computing the lowest cost match make this possible. Both the BV scheme and the ABV scheme solve a more general packet classification problem.

B.2.2.4 Decision Graphs

Decision graphs used to classify packets consist of nodes that represent a test of a variable's value. The branches of the node represent the path taken depending on the current value of the variable. They may represent one value or a range of values. They differ from the types of graphs discussed previously in that a node may have more than two children and can thus have complex expressions at the nodes, rather than just Boolean predicates. In one implementation of such a graph, the graph is constructed in a number of layers corresponding to the dimensionality of the filter, so each layer corresponds to the test of one field in the packet header [D. Decasper and Plattner(1998)]. Because

of this, the depth of the graph is same as the number of fields filtered on and thus the lookup time is roughly linear in number of fields and independent of the number of rules. However, no results are given regarding the memory requirements of this scheme.

B.2.2.5 Hierarchical Intelligent Cuttings

Another technique called HiCuts [Gupta and McKeown(1999b)] builds a decision graph in a similar way, except that each leaf stores a number of filter rules rather than just the single matching rule or rule number. The lookup algorithm traverses the decision graph, which is typically quite shallow, to a leaf node and then performs a linear search through the list of rules stored at the leaf node. The depth, shape and decisions to be made at each node greatly affect the performance of the resulting decision graph and so the algorithm is heavily guided by heuristics. During the construction of the graph, the partition that leads to the uniform distribution of the rules across the nodes is chosen. When the number of rules at a node drops below a certain threshold, the node is not partitioned any further. Experimental work on access lists ranging from 100 to 1700 rules in size with 4 fields and the above threshold set to 8, measured a worst case lookup requiring 12 memory accesses and a linear search on 8 rules. No analytical results are given.

Gupta and McKeown [Gupta and McKeown(1999b)] uses heuristics to solve k-dimensional packet classification problem. Their approach focuses on the practical implementation of classification with real-life filter database. The approach, called HiCuts (hierarchical intelligent cuttings), attempts to partition the search space in each dimension, guided by simple heuristics that exploit the structure of filter database.

The HiCuts algorithm builds a decision-tree data structure by carefully preprocessing the filter database. Each time a packet arrives, the classification algorithm traverses the decision tree to find a leaf node, which stores a small number of rules. A linear search of these rules produces the matching filter.

Figure B.6 illustrates the geometrical representation of a two-dimensional filter database. Figure B.7 shows a possible tree for the database.

All these heuristics are combined to create the best decision tree for the filter database and tuning parameters for these heuristics would possibly create different trees.

B.2.2.6 Binary Decision Diagram

Hazelhurst [Hazelhurst(1999)] presents the idea of transforming firewall packet filters into Boolean expressions that are represented as BDDs. The paper describes an algorithm for transforming a CISCO firewall filter into a BDD, including the handling of issues

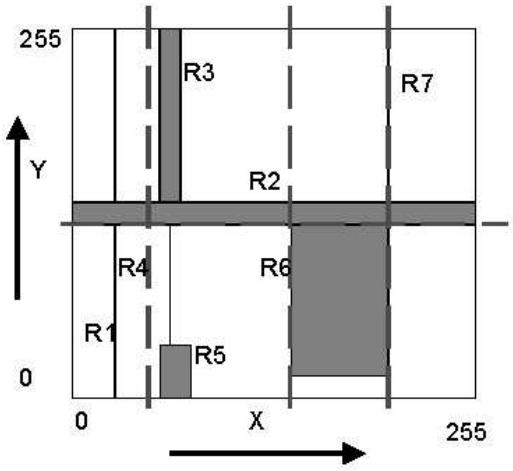

FIGURE B.6: Geometrical representation of seven filters

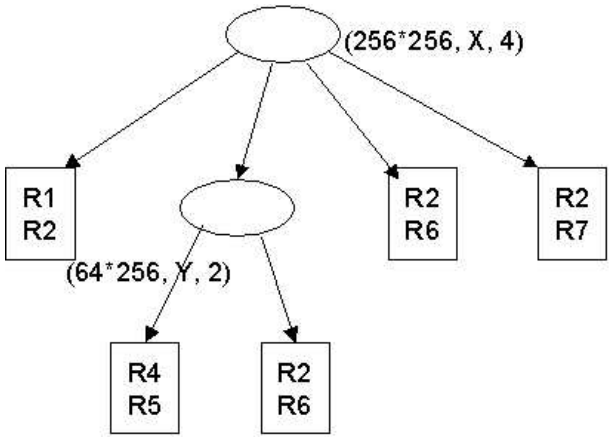

FIGURE B.7: A possible tree for filters in Figure B.6

with overlapping rules. The main use of BDDs in this paper is for a tool that can be used analysing and test filters. A later paper by Hazelhurst et al.[S. Hazelhurst and Sinnappan(2000)] focus on using the BDD structures for performing packet classification. The conclusion is that BDDs can improve the lookup latency on systems using dedicated hardware such as FPGAs, while they do not perform well on generic CPUs.

Attar and Hayelhurst [Attar and Hazelhurst(2002)] use N-ary decision diagrams for improving the lookup performance. The experimental results show that the lookup time can be significantly improved by using this method, however at the price of increased memory usage. Furthermore, the idea of using MTBDDs to handle the more general packet classification is suggested.

Most software approaches involve improving the data representations for rule storage and the rule search algorithms. Software approaches suffer from two inherent limitations. Firstly, the filter runs sequentially on a general-purpose processor. Hence, there is a limit to performance no matter how well the data structures and algorithms are optimised. Secondly, software solutions suffer from high overheads. For instance, several layers

of hardware and software have to be traversed during filtering: the packet must be transferred from the physical cable to the network card, through the protocol stack and to the filtering software, and back to the network again, traversing many buffers and buses in the process. Other overheads result from operating system tasks such as interrupt handling. Hardware approaches, attempt to overcome these limitations.

B.2.3 Hardware-based Classification

A simple hardware approach is to use multiple general purpose processors to filter several packets at a time [Benecke(1999)]. However, this only increases filtering throughput, and does not decrease the lookup time itself. Furthermore, using multiple processors would increase the cost of the filter.

Many manufacturers have used custom hardware to reduce lookup latency. Their approaches essentially consist of embedding the lookup algorithms in application specific integrated circuits (ASICs). For example, Fore Systems Inc. produces Ethernet switches that execute a microcode version of CheckPoint's Firewall-1 software on ASICs [3]. This allows aggregate data rates of up to 20 Gbits per second, as opposed to the 50- to 100 Mbits per second speeds of other filters [Newman(1999)]. Juniper Networks also produces the Internet Processor II ASIC that reportedly has filtering rates of 20 million packets per second, whereas software filters reportedly approach a limit of 200,000 packets per second [Lear(2000)]. Unfortunately, these high speeds are matched by the high cost of the filters, making ASIC-based packet filters a viable option only for large corporations.

Another approach that has been tried, is the use of content addressable memory (CAM) [Neale(1999)]. CAM allows the contents of memory locations to be searched in parallel given an input key value. Hence, if the rulebase is stored in CAM, a rule whose condition matches the fields in a given packet can quickly be found. However, CAMs are too small, too expensive, and consume too much power for classification on many fields [Gupta and McKeown(1999a)].

FPGAs have also been used in packet filtering applications. McHenry et al.[John T. McHenry and Cocks(1997)] used an FPGA for packet filtering in ATM networks. Their approach involves using a combination of a dedicated personal computer (the firewall control processor, FCP) and an FPGA-based firewall inline processor (FIP). The FCP performs the usual rule lookup and authentication associated with a firewall, and the FIP acts as ATM cell forwarder between the transmitter and the sender. When an ATM transmission begins, the FCP authenticates the connection and if it is authorised, it instructs the FIP to forward the ATM cells that make up the transmission. If the transmission is not authorised, the FCP

[3] Hardware developers do not release details of the internal operation of the ASICs, possibly for commercial reasons. Hence, it is not clear how the ASICs execute the microcode, for instance, whether in parallel or sequentially.

instruct the FIP to discard the cells and hence the transmission does not take place. Due to the way ATM works, only the first cell needs to authenticated by the FCP, after that, the other cells can pass through the FIP without intervention from the FCP.

Lakshman and Stiliadis [Lakshman and Stiliadis(1998)], and Bailey et al.[M. L. Bailey and Sarkar(1994)] also describe FPGA implementations. In these approaches, some representation of the rulebase is stored in RAM, either within the FPGA [M. L. Bailey and Sarkar(1994)] or on RAM chips [Lakshman and Stiliadis(1998)] and an FPGA is used to implement an algorithm to perform rule lookup on the RAM.

Appendix C

Virus Detection Result by Janus *VirusDetector*

All the test results of *VirusDetector* are listed here. Tested files were either Win9x or Win32 executable files. The tables are listed as encrypted parasitic viruses, polymorphic viruses and parasitic viruses.

TABLE C.1: Win9x Encrypted Parasitic Virus Detection Result by *VirusDetector*
Encrypted Win9x Virus Total Number: 30, Error number: 1, False negative: 0.03 (3%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Bumble.1736	8192	3-1-04	O	Bumble.1738	8192	11-22-03	O
Iced.1344	44032	9-6-01	O	Iced.1376	8192	11-22-03	O
Iced.1412	8192	11-22-03	O	Iced.1617	8192	3-1-04	O
Iced.2112	8192	3-1-04	O	Mad.2736.a	32768	10-4-04	O
Mad.2736.b	32768	10-4-04	O	Mad.2736.c	32768	1-7-02	O
Mad.2736.d	32768	1-7-02	O	Mad.2806	32768	1-19-98	O
Nathan.3276	7372	9-2-02	O	Nathan.3520.a	12288	3-10-04	O
Nathan.3520.b	16384	9-9-01	O	Nathan.3792	185552	10-23-99	O
Obsolete.1419	5003	3-1-04	O	PoshKill.1398	8192	4-21-01	O
PoshKill.1406	8192	3-17-03	O	PoshKill.1426	8192	8-20-01	O
PoshKill.1445.a	8192	3-10-04	O	PoshKill.1445.b	8192	11-22-03	O
Priest.1419	4096	8-9-99	O	Priest.1454	4096	3-1-04	O
Priest.1478	9728	6-10-00	X	Priest.1486	9728	10-4-01	O
Priest.1495	9728	8-18-01	O	Priest.1521	9728	3-1-04	O
Shoerec	321536	8-12-01	O	Tip.2475	10752	11-22-03	O
Voodoo.1537	61441	11-22-03	O	Werther.1224	6344	12-31-01	O

TABLE C.2: Win32 Encrypted Parasitic Virus Detection Result by *VirusDetector*
Encrypted Win32 Virus Total Number: 30, Error number: 4, False negative: 0.13 (13%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Ditto.1488	6096	3-1-04	O	Ditto.1492	12288	11-22-03	O
Ditto.1539	8192	10-1-00	O	Gloria.2820	16384	11-22-03	X
Gloria.2928	16384	3-1-04	O	Gloria.2963	12288	10-1-00	O
Idele.2104	8192	7-8-03	O	Idele.2108	8192	3-1-04	O
Idele.2160	8192	11-12-03	O	IhSix.3048	8192	11-22-03	O
Infinite.1661	8192	3-1-04	O	Levi.2961	12288	5-16-01	O
Levi.3040	7188	11-22-03	O	Levi.3090	12288	11-22-03	O
Levi.3137	35941	11-22-03	O	Levi.3205	12288	3-1-04	X
Levi.3240	16384	8-17-02	O	Levi.3244	16384	11-22-03	X
Levi.3432	16384	11-22-03	O	Mix.1852	4096	5-30-00	O
Niko.5178	65611	11-22-03	X	Santana.1104	81920	12-4-01	O
Savior.1680	8192	1-8-01	O	Savior.1696	12288	5-18-01	O
Savior.1740	12288	3-28-02	O	Savior.1828	20480	8-6-01	O
Savior.1832	12288	3-1-04	O	Savior.1904	12288	12-4-01	O
Undertaker.4887	12288	11-22-03	O	Undertaker.5036.a	12288	11-22-03	O

TABLE C.3: Win9x Polymorphic Virus Detection Result by *VirusDetector*
Win9x Polymorphic Virus Total Number: 15, Error number: 2, False negative: 0.13 (13%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Begemot	8192	3-1-04	O	Darkmil.5086	12288	3-1-04	X
Darkmil.5090	74210	7-2-03	O	Fiasko.2500.a	8192	11-22-03	O
Fiasko.2500.b	12935	3-1-04	O	Fiasko.2508	8192	3-1-04	O
Invir.7051	9728	3-1-04	O	Luna.2636	8192	4-24-02	O
Luna.2757.a	62213	3-10-04	O	Luna.2757.b	12288	1-1-80	O
Marburg.a	493789	3-10-04	O	Marburg.b	28381	11-22-03	X
Matrix.3597	35916	2-14-03	O	Merinos.1763	9216	3-1-04	O
Merinos.1849	8192	11-22-03	O				

TABLE C.4: Win32 Polymorphic Virus Detection Result by *VirusDetector*

Encrypted Win32 Virus Total Number: 50, Error number: 21, False negative: 0.42 (42%)

Date indicates the virus's detected & caught date in form MM-DD-YY.

Virus names are also the names of the test files. Size unit is byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Andras.7300	14238	7-27-02	O	AOC.2044	8192	11-28-99	O
AOC.2045	8192	11-26-99	O	AOC.3657	16384	3-1-04	O
AOC.3833	16384	3-1-04	O	AOC.3860	20480	2-10-03	X
AOC.3864	20480	2-10-03	O	Champ	12288	2-10-03	O
Champ.5430	12288	10-6-02	O	Champ.5464	12288	2-10-03	O
Champ.5477	12288	10-6-02	O	Champ.5495	6144	2-10-03	X
Champ.5521	12288	2-10-03	X	Champ.5536	12288	2-10-03	X
Champ.5714	12288	2-10-03	O	Champ.5722	16384	2-10-03	X
Chop.3808	64049	8-29-01	X	Crypto	49152	3-1-04	X
Crypto.a	28672	11-22-03	X	Crypto.b	32768	11-22-03	O
Crypto.c	32768	11-22-03	O	Driller	94208	3-1-04	O
Harrier	108544	11-22-03	X	Hatred.a	16384	3-10-04	O
Hatred.d	16384	10-29-02	O	Kriz.3660	415232	7-27-02	X
Kriz.3740	764928	10-4-04	X	Kriz.3863	475136	10-4-04	X
Kriz.4029	12288	3-1-04	O	Kriz.4037	12288	8-19-01	O
Kriz.4050	479232	11-22-03	X	Kriz.4057	12288	8-19-01	X
Kriz.4075	12288	11-22-03	O	Kriz.4099	12288	11-22-03	X
Kriz.4233	8192	7-15-01	X	Kriz.4271	57344	10-4-04	O
Prizm.4428	8704	9-4-02	X	RainSong.3874	8192	11-22-03	O
RainSong.3891	61509	3-1-04	O	RainSong.3910	8192	12-5-01	X
RainSong.3956	12288	10-4-04	X	RainSong.4198	8192	3-12-02	O
RainSong.4266	12288	10-4-04	X	Thorin.11932	16384	3-1-04	O
Thorin.b	16384	3-1-04	O	Thorin.c	16384	10-23-99	O
Thorin.d	16384	7-14-99	O	Thorin.e	16384	10-23-99	O
Vampiro.7018	18432	3-1-04	X	Vampiro.a	16896	3-10-04	O

TABLE C.5: Win9x Virus Detection Result by *VirusDetector*

Win9x Virus Total Number: 291, Error number: 26, False negative: 0.09 (approx. 9%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Altar.797	8192	6-19-99	O	Altar.884	4096	10-4-02	O
Altar.910	4096	10-24-99	O	Antic.695	7863	9-2-02	O
Anxiety.1358	152778	3-1-04	O	Anxiety.1397	49736	3-1-04	O
Anxiety.1399	8192	3-21-98	O	Anxiety.1399.b	49736	3-1-04	O
Anxiety.1422	5684	11-22-03	O	Anxiety.1451	29213	2-21-01	O
Anxiety.1486	8192	1-24-98	O	Anxiety.1517	8192	11-22-03	O
Anxiety.1586	42590	11-22-03	O	Anxiety.1596	49736	3-11-98	O
Anxiety.1823	8192	3-1-04	O	Anxiety.1823.b	6196	7-2-03	O
Anxiety.2471	8192	11-22-03	O	Apop.1086	8192	7-17-02	O
Argos.310	4096	9-2-02	O	Argos.328	4096	3-1-04	O
Argos.335	4096	9-24-02	O	Argos.402	4096	6-30-99	O
Arianne.1022.a	4606	3-10-04	O	Arianne.1022.b	94112	11-22-03	O
Arianne.1052	8192	5-16-02	O	Atom.4790	64182	5-8-02	X
Babylonia.11036	33734	3-1-04	X	Babylonia.attach	5984	5-16-02	O
Babylonia.Plugin.Dropper	12606	11-22-03	X	Babylonia.Plugin.Greetz	621	11-22-03	X
Babylonia.Plugin.IrcWorm	1707	11-22-03	O	Babylonia.Plugin.Poll	1041	11-22-03	O
Begemot	8192	3-1-04	O	BlackBat.2615	8192	5-31-01	O
BlackBat.2787	8192	11-22-03	O	BlackBat.2795	8192	11-22-03	O
BlackBat.2840	8192	5-27-01	X	BlackBat.2841.a	8192	3-10-04	O
BlackBat.2841.b	8192	5-31-01	X	BlackBat.2988	8192	11-22-03	O
Bodgy.3230	97438	4-15-03	X	Bonk.1232	19632	11-22-03	O
Bonk.1243	9460	3-1-04	O	Boza.2220	24576	11-22-03	X
Boza.A	12408	9-2-99	O	Boza.C	16384	9-10-99	O
Boza.a	12408	3-17-96	O	Boza.b	7994	11-22-03	O
Boza.c	16384	3-1-04	O	Boza.d	16384	11-22-03	O
Boza.e	16384	11-22-03	O	Bumble.1736	8192	3-1-04	O
Bumble.1738	8192	11-22-03	O	Butool.910	14222	9-2-02	O
Buzum.1828	6310	3-1-04	O	ByteSV.Thorn.886	20480	11-22-03	O
Caw.1262	205550	11-22-03	O	Caw.1335	5943	3-1-04	O
Caw.1416	55964	11-22-03	O	Caw.1419	54667	11-22-03	X
Caw.1457	6065	11-22-03	O	Caw.1458	8192	10-5-02	O
Caw.1525	24576	11-22-03	O	Caw.1531	8192	11-2-01	O
Caw.1557	180224	12-31-00	O	Chimera.1542	35846	11-22-03	O
CIH	19536	7-8-02	O	CIH.v1.2	19536	9-12-99	O
CIH.v1.3	36864	9-3-99	O	CIH.v1.4	4608	9-22-99	O
CIH.1003.b	4608	3-1-04	O	CIH.1010.b	37394	11-22-03	O
CIH.1016	34304	9-2-02	O	CIH.1019.c	4896	3-1-04	O
CIH.1024	1553	3-1-04	O	CIH.1026	1555	3-1-04	O
CIH.1031	4096	9-2-02	O	CIH.1035	1564	3-1-04	O
CIH.1040	159744	3-1-04	O	CIH.1042	53248	3-1-04	O
CIH.1048	20480	9-2-02	O	CIH.1049	65536	9-2-02	O
CIH.1103	2144	11-22-03	O	CIH.1106	153088	11-27-02	O
CIH.1122	1651	9-2-02	O	CIH.1129	1658	3-1-04	X
CIH.1142	16384	6-19-98	O	CIH.1230	1759	3-1-04	O
CIH.1262	1778	3-1-04	O	CIH.1297	1826	11-22-03	O
CIH.1363	59392	3-1-04	X	CIH.2563	24576	9-2-02	X
CIH.2690	94208	9-2-02	X	CIH.816.a	59392	9-2-02	X

TABLE C.6: Win9x Virus Detection Result by *VirusDetector* (Cont.)
Win9x Virus Total Number: 291, Error number: 26, False negative: 0.09 (approx. 9%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
CIH.816.b	5120	9-2-02	O	CIH.862	1903	5-10-02	O
CIH.876	20480	9-2-02	X	CIH.913	20480	9-2-02	O
CIH.937	20480	9-2-02	O	CIH.973	1502	3-1-04	O
CIH.corrupted	188478	1-10-02	O	CIH.dam	1090318	1-1-03	X
CIH-II.776	53248	9-2-02	O	CIH-II.882	20480	9-29-02	O
CIH.intended	2426	8-31-99	O	CIH-Killer.1373	9053	3-1-04	O
CIH.src	301056	3-1-04	X	Companion.4096	4096	3-1-04	O
Croman	16384	9-2-02	X	Dado	332817	3-1-04	X
Darkmil.5086	12288	3-1-04	X	Darkmil.5090	74210	7-2-03	O
DarkSide.1105	8192	4-6-02	O	DarkSide.1371	8192	3-1-04	O
DarkSide.1491	9728	10-4-04	O	Dead.1086	8192	9-29-02	O
Dead.4172	11392	9-2-02	O	Dead.4316	16796	5-16-02	O
Dead.4388	11608	9-2-02	O	Demo.8192	8192	11-22-03	O
Dodo.1022	8192	9-2-02	O	Dupator.1503	110592	1-29-04	O
Eak	103424	12-19-02	X	Esmeralda.807	4955	7-15-01	O
Etymo.1308	7168	3-1-04	O	Evil.953.a	60345	11-22-03	X
Evil.953.b	4096	9-24-02	O	Evil.962	8192	3-1-04	O
Evil.962.b	35266	11-22-03	O	Evil.962.c	4096	8-26-01	O
Federal	8192	9-2-02	O	Fiasko.2500.a	8192	11-22-03	O
Fiasko.2500.b	12935	3-1-04	O	Fiasko.2508	8192	3-1-04	O
Filth.1030	4096	3-1-04	O	Flee.835	8192	11-22-03	O
Fono.15327	24064	3-1-04	X	Fono.Trojan	263	5-16-02	O
Frone.864	69632	5-16-02	X	Frone.951	8192	9-2-02	O
FYS.1728	8192	7-15-01	O	Gara.640	8192	11-22-03	O
Gara.842.a	8192	3-10-04	O	Gara.842.b	8192	11-22-03	O
Gara.917	8192	12-4-01	O	Harry.a	8192	3-10-04	O
Harry.b	8192	6-2-97	O	Hooy.8192	32768	3-28-00	O
Horn.1851	6322	3-1-04	O	Horn.1862	6334	11-22-03	O
Horn.2223	6695	11-22-03	O	Horn.2245	6719	3-1-04	O
HPS.5124	26563	3-1-04	X	I13.a	8192	11-22-03	O
I13.b	12288	3-1-04	O	I13.c	8192	3-1-04	O
I13.d	12288	11-22-03	X	I13.e	8192	11-22-03	O
I13.f	8192	10-4-02	O	Iced.1344	44032	9-6-01	O
Iced.1376	8192	11-22-03	O	Iced.1412	8192	11-22-03	O
Iced.1617	8192	3-1-04	O	Iced.2112	8192	3-1-04	O
Icer.541	4096	11-22-03	O	Icer.619	192512	1-13-04	O
ILMX.1291	53248	3-1-04	O	Invir.7051	9728	3-1-04	O
Jacky.1440	4646	3-1-04	O	Jacky.1443	8192	11-22-03	O
Javel.512	1529	3-1-04	O	Julus.1904.a	8192	11-22-03	O
Julus.1904.b	8192	10-4-04	O	Julus.1929.a	8192	11-22-03	O
Julus.1929.b	8192	1-23-03	O	Julus.2702.a	12288	11-22-03	O
Julus.2702.b	8192	8-29-01	O	Julus.2777	12288	11-22-03	O
K32.1012	5108	11-22-03	O	K32.2929	8192	10-4-02	O
K32.3030	9174	3-1-04	O	K32.Roma.2929	9643	12-9-00	O
Kaze	20480	8-8-02	O	Kurgan.10240	14336	9-2-02	O
Lizard.1967	7099	3-1-04	O	Lizard.2381	2957	3-1-04	O
Lizard.2869	3715	3-1-04	O	Lizard.5150	5150	10-4-04	O

TABLE C.7: Win9x Virus Detection Result by *VirusDetector* (Cont.)
Win9x Virus Total Number: 291, Error number: 26, False negative: 0.09 (approx. 9%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Lorez.1766.a	8192	3-10-04	O	Lorez.1766.b	8192	6-28-01	O
LoveSong.998	61440	12-4-01	O	Lud.Hill.401	8192	3-1-04	O
Lud.Jadis.3567	9216	3-1-04	O	Lud.Jadis.3579	9216	11-22-03	O
Lud.Jez.676	8192	3-1-04	O	Lud.Jez.682	8192	11-22-03	O
Lud.Yel.1886	22141	10-4-04	O	Luna.2636	8192	4-24-02	O
Luna.2757.a	62213	3-10-04	O	Luna.2757.b	12288	1-1-80	O
Mad.2736.a	32768	10-4-04	O	Mad.2736.b	32768	10-4-04	O
Mad.2736.c	32768	1-7-02	O	Mad.2736.d	32768	1-7-02	O
Mad.2806	32768	1-19-98	O	Marburg.a	493789	3-10-04	O
Marburg.b	28381	11-22-03	X	MarkJ.826	8192	3-1-04	O
Matrix.3597	35916	2-14-03	O	Memorial	35515	2-7-98	O
Merinos.1763	9216	3-1-04	O	Merinos.1849	8192	11-22-03	O
MMort.1335	8192	11-22-03	O	MMort.1340	8192	11-22-03	O
MMort.1348	8192	3-1-04	O	MMort.1366	8192	10-23-99	O
Molly.680	8192	5-16-02	O	Molly.722	8192	3-1-04	O
MrKlunky.a	6943	3-10-04	O	MrKlunky.b	6779	7-2-03	X
MSpawn.4608	8897	11-22-03	O	Murkry.383	4096	5-15-02	O
Murkry.398.a	59392	11-22-03	O	Murkry.398.b	4096	9-24-02	O
Murkry.399	4096	3-1-04	O	Murkry.441	26624	9-24-02	O
Nathan.3276	7372	9-2-02	O	Nathan.3520.a	12288	3-10-04	O
Nathan.3520.b	16384	9-9-01	O	Nathan.3792	185552	10-23-99	O
Noise.414	57344	3-1-04	O	Obsolete.1419	5003	3-1-04	O
Onerin.371	4096	1-9-02	O	Onerin.383	4096	1-9-02	O
Opa.1103	45056	5-8-01	O	Opa.1149	45056	7-15-01	O
Padania.1335	8192	3-1-04	O	Paik.1908	8192	5-14-01	O
PoshKill.1398	8192	4-21-01	O	PoshKill.1406	8192	3-17-03	O
PoshKill.1426	8192	8-20-01	O	PoshKill.1445.a	8192	3-10-04	O
PoshKill.1445.b	8192	11-22-03	O	Powerful.1592	6144	3-1-04	X
Powerful.1773	6144	3-1-04	O	Powerful.1901	12288	7-12-01	O
Priest.1419	4096	8-9-99	O	Priest.1454	4096	3-1-04	O
Priest.1478	9728	6-10-00	X	Priest.1486	9728	10-4-01	O
Priest.1495	9728	8-18-01	O	Priest.1521	9728	3-1-04	O
Prizm.4428	8704	9-4-02	X	Puma.1024	4096	3-1-04	O
Regix.4096.a	8192	3-10-04	O	Sanat.3151	16384	3-1-04	X
Shoerec	321536	8-12-01	O	Sign.2028	8192	1-8-01	O
Smash.10262	16384	8-16-01	X	SST.952	4096	3-1-04	O
Tecata.1761	66029	10-4-04	O	Tenrobot.b	49152	5-14-03	X
Tip.2475	10752	11-22-03	O	Titanic.3214	7822	8-16-01	O
Uwaga.3237	8192	11-22-03	O	Vivic	8192	8-17-02	O
Voodoo.1537	61441	11-22-03	O	Werther.1224	6344	12-31-01	O
Whal.a	8192	1-24-01	O	Whyg.1193	8192	6-24-01	O
Yabran.3132	4608	3-1-04	O	Yobe	20480	3-1-04	O
Youd.1388	8192	11-22-03	O	Yoyo.653	4096	3-1-04	O
Zerg.3849	8192	3-1-04	O	Zofo.848	20480	3-1-04	O
Zoual	147456	10-18-02	X				

TABLE C.8: Win32 Virus Detection Result by *VirusDetector*.

Win32 Virus Total Number: 499
Error number: 103, False negative: 0.2064 (approx. 21%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is Byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Adson.1559	8192	7-26-02	O	Adson.1734	20480	11-22-03	O
Aidlot	8192	10-4-04	O	Akez	32768	4-27-02	O
Aliser.7825	12288	3-6-03	O	Aliser.7897	8192	6-8-03	O
Aliser.8381	8192	6-8-03	O	Alma.2414	10606	3-1-04	O
Alma.37195	45387	11-22-03	X	Alma.37274	40960	4-19-02	X
Alma.5319	13511	3-1-04	X	Andras.7300	14238	7-27-02	O
AOC.2044	8192	11-28-99	O	AOC.2045	8192	11-26-99	O
AOC.3657	16384	3-1-04	O	AOC.3833	16384	3-1-04	O
AOC.3860	20480	2-10-03	X	AOC.3864	20480	2-10-03	O
Apathy.5378	8192	3-1-04	O	Apparition	96239	12-22-99	O
Apparition.a	542861	3-10-04	X	Apparition.b	167707	6-5-98	X
Arianne.1052	6684	3-11-02	O	Aris	331785	3-1-04	X
Arrow.a	2048	10-5-04	O	Artelad.2173	23040	12-31-00	O
Asorl.a	32269	3-10-04	X	AutoWorm.3072	3072	3-1-04	O
Awfull.2376	3072	9-9-03	O	Awfull.3571	4096	3-1-04	O
Bakaver.a	24576	10-6-03	O	Banaw.2157	8192	11-22-03	O
Barum.1536	5632	8-31-02	O	Bee	24576	3-1-04	O
Beef.2110	57344	3-1-04	O	Belial.2537	8192	11-22-03	O
Belial.2609	254513	3-9-02	X	Belial.a	4096	3-10-04	O
Belial.b	4096	2-23-02	O	Belial.c	4096	11-22-03	O
Belial.d	4096	9-24-02	O	Belod.a	8192	3-12-02	O
Belod.b	8192	8-21-02	O	Belod.c	8192	3-14-02	O
Bender.1363	3584	12-31-01	O	Bika.1906	8192	12-31-01	O
BingHe	296643	10-11-02	X	Blackcat.2537	8192	9-9-03	O
Blakan.2016	8192	12-31-01	O	Blateroz	8192	9-2-02	O
Blueballs.4117	16384	11-22-03	X	Bobep	8192	8-25-03	O
Bogus.4096	38400	10-13-99	O	Bolzano.2122	36864	2-10-03	O
Bolzano.2664	15135	2-10-03	O	Bolzano.2676	15183	2-10-03	O
Bolzano.2716	13521	2-10-03	O	Bolzano.3100	15277	2-10-03	O
Bolzano.3120	15331	2-10-03	O	Bolzano.3148	15373	2-10-03	O
Bolzano.3164	15409	2-10-03	O	Bolzano.3192	15457	3-1-04	O
Bolzano.3628	16095	3-1-04	O	Bolzano.3904	16251	2-10-03	O
Bolzano.5572	28237	2-10-03	O	Butter	96665	9-2-02	X
Cabanas.a	7171	5-7-04	X	Cabanas.b	7171	3-1-04	O
Cabanas.Debug	95748	10-4-04	O	Cabanas.e	16384	7-8-03	O
Cabanas.MsgBox	39996	10-4-04	X	Cabanas.Release	49152	1-26-99	O
CabInfector	4096	3-1-04	O	Cecile	28672	12-31-01	X
Cefet.3157	7253	9-2-02	O	Cerebrus.1482	8192	3-1-04	O
Champ	12288	2-10-03	O	Champ.5430	12288	10-6-02	O
Champ.5464	12288	2-10-03	O	Champ.5477	12288	10-6-02	O
Champ.5495	6144	2-10-03	X	Champ.5521	12288	2-10-03	X
Champ.5536	12288	2-10-03	X	Champ.5714	12288	2-10-03	O
Champ.5722	16384	2-10-03	X	Chatter	22528	1-13-03	O
Chop.3808	64049	8-29-01	X	Cornad	4096	6-22-03	O
Crosser	102400	12-6-03	X	Crypto	49152	3-1-04	X

TABLE C.9: Win32 Virus Detection Result by *VirusDetector* (Cont.),
Win32 Virus Total Number: 499
Error number: 103, False negative: 0.2064 (approx. 21%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is Byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Crypto.a	28672	11-22-03	X	Crypto.b	32768	11-22-03	O
Crypto.c	32768	11-22-03	O	Damm.1624	24576	3-1-04	O
Damm.1628	4096	5-16-02	O	Damm.1647.a	12288	11-22-03	O
Datus	105472	5-16-02	X	Delikon	16384	1-6-04	O
Devir	24576	10-4-04	O	Dictator.2304	10496	11-22-03	X
Dislex	135239	3-1-04	X	Ditex	212992	4-17-02	O
Ditto.1488	6096	3-1-04	O	Ditto.1492	12288	11-22-03	O
Ditto.1539	8192	10-1-00	O	Donny.a	8192	3-10-04	O
Donut	12800	3-1-04	O	Dream.4916	69632	3-1-04	O
Driller	94208	3-1-04	O	Drivalon.1876	3072	7-8-03	O
Dudra.5632	12288	7-7-01	O	Eclipse.a	8192	3-10-04	O
Eclipse.b	6644	3-15-01	O	Eclipse.c	8192	8-18-99	O
Egolet.a	4096	3-10-04	O	Egolet.b	4096	7-7-02	O
Elerad	8192	2-2-02	X	Emotion.a	4608	3-10-04	O
Emotion.b	8192	11-30-00	O	Emotion.c	8192	2-10-03	O
Emotion.d	8192	2-10-03	O	Emotion.gen	8192	9-20-01	O
Enar	89088	3-1-04	O	Enumiacs.6656	6656	11-22-03	O
Enumiacs.8192.a	274	10-4-04	O	Fighter.a	6656	11-22-03	O
Fighter.b	8192	1-29-04	X	Flechal	69632	3-1-04	O
Fosforo.a	8192	3-10-04	O	Fosforo.b	8192	10-24-02	O
Fosforo.c	8192	12-31-01	X	Fosforo.d	8192	3-5-03	X
Freebid	14066	8-27-02	O	FunLove.4070	69635	3-7-04	O
Gaybar	55493	2-9-04	O	gen	8192	9-3-02	O
Genu.a	8192	11-22-03	O	Genu.b	8192	11-22-03	O
Genu.c	5619	8-18-02	O	Genu.d	8192	7-26-02	O
Ghost.1667	8192	3-1-04	O	Ginra.3334	8966	8-31-02	O
Ginra.3413	8192	9-2-02	O	Ginra.3570	8192	5-18-02	O
Ginra.3657	8192	10-4-04	O	Ginseng	4096	8-24-02	O
Giri.4919	185143	11-22-03	O	Giri.4970	13162	3-9-02	O
Giri.5209	12288	3-28-01	X	Gloria.2820	16384	11-22-03	X
Gloria.2928	16384	3-1-04	O	Gloria.2963	12288	10-1-00	O
Glyn	8192	3-1-04	O	Gobi.a	4096	10-23-02	O
Godog	12288	3-1-04	O	Golsys.14292	55252	8-30-02	X
Grenp.2804	4608	11-22-03	O	Halen.2593	8192	3-1-04	O
Halen.2618	22743	7-27-02	O	Halen.2619	8192	7-12-02	O
Haless.1127	31744	10-4-04	X	Harrier	108544	11-22-03	X
Hatred.a	16384	3-10-04	O	Hatred.d	16384	10-29-02	O
Hawey	5595	7-12-03	O	Heretic.1986	8192	3-1-04	O
Hezhi	152064	9-2-02	O	Hidrag.a	36352	8-24-01	X
Highway.a	8192	11-22-03	O	Highway.b	48177	3-9-02	O
HIV	175	2-10-03	X	HIV.6340	12288	11-22-03	O
HIV.6382	12288	11-22-03	O	HIV.6386	12288	10-4-04	X
HIV.6680	12288	3-1-04	X	HLL.Fugo	55808	7-6-04	X

TABLE C.10: Win32 Virus Detection Result by *VirusDetector* (Cont.),
Win32 Virus Total Number: 499
Error number: 103, False negative: 0.2064 (approx. 21%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is Byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
HLLP.BadBy	329728	11-22-03	X	HLLP.Bora.11264	11264	5-8-02	X
HLLP.Clay	60416	11-22-03	O	HLLP.Delvi	46080	3-11-02	X
HLLP.Famer	22528	1-18-03	X	HLLP.Freefall	36352	6-15-02	X
HLLP.Gezad	28672	7-8-03	O	HLLP.Givin	34820	10-4-04	O
HLLP.Gosus	69590	8-17-02	O	HLLP.Gotem	22016	11-25-02	X
HLLP.Hetis	38400	2-27-02	O	HLLP.Imel	34816	8-30-02	O
HLLP.Karabah	190988	11-22-03	O	HLLP.Kiro	79632	4-9-04	X
HLLP.Lassa.40960	40960	5-8-02	O	HLLP.Mincer	173315	4-19-02	X
HLLP.MTV	68096	11-22-03	X	HLLP.Nilob	24576	7-14-00	O
HLLP.Pres	124936	9-2-02	X	HLLP.Semisoft	59904	12-4-99	O
HLLP.Shodi.c	98318	4-9-04	X	HLLP.Sloc	104448	8-31-02	O
HLLP.Sneak	34816	3-1-04	O	HLLP.Thembe	130322	3-1-04	X
HLLP.Unzi	24576	3-25-02	O	HLLP.Winfig	33280	11-22-03	O
HLLP.Yai	341211	11-14-99	X	Htrip.a	8192	12-4-01	O
Htrip.b	8192	11-22-03	O	Htrip.c	8192	12-4-01	O
Idele.2104	8192	7-8-03	O	Idele.2108	8192	3-1-04	O
Idele.2160	8192	11-12-03	O	IhSix.3048	8192	11-22-03	O
IKX	4096	3-1-04	O	Infinite.1661	8192	3-1-04	O
Infis.4608	4608	3-1-04	O	Initx	210432	10-4-04	X
Insom.1972.a	5120	4-23-04	O	InvictusDLL.099	4096	11-22-03	O
InvictusDLL.102	8704	9-15-01	X	InvictusDLL.a	8192	3-10-04	X
InvictusDLL.b	8192	8-17-01	X	InvictusDLL.c	8704	9-10-01	X
InvictusDLL.d	56466	5-5-99	X	Ipamor.a	65536	3-10-04	X
Ipamor.c	38913	5-15-03	X	Ipamor.d	35840	5-15-03	X
Ivaz	4096	11-22-03	O	Jater	4096	3-1-04	O
Jethro.5657	17433	3-1-04	O	Junkcomp	65536	12-30-02	X
Kala.7620	65536	12-30-01	X	Kanban.a	3072	3-10-04	O
Keisan.a	8192	6-2-03	O	Keisan.b	8192	6-2-03	O
Keisan.c	8192	6-2-03	O	Keisan.d	8192	6-2-03	O
Keisan.e	8192	6-2-03	O	Ketan	4096	3-1-04	O
Kiltex	155648	3-1-04	O	Klinge	8192	3-1-04	O
KME	36864	3-1-04	O	KMKY	24576	8-6-01	X
Knight.2350	6958	12-4-01	O	Koru	65536	5-15-98	O
Kriz.3660	415232	7-27-02	X	Kriz.3740	764928	10-4-04	X
Kriz.3863	475136	10-4-04	X	Kriz.4029	12288	3-1-04	O
Kriz.4037	12288	8-19-01	O	Kriz.4050	479232	11-22-03	X
Kriz.4057	12288	8-19-01	X	Kriz.4075	12288	11-22-03	O
Kriz.4099	12288	11-22-03	X	Kriz.4233	8192	7-15-01	X
Kriz.4271	57344	10-4-04	O	Kuto.2058	10250	8-31-02	X
Lad.1916	61440	9-2-02	X	Ladmar.2004	57344	9-1-02	O
Lamebyte	8192	4-30-03	O	Lames.4096	8192	1-3-98	O
Lamewin.1751	3584	8-3-02	O	Lamewin.1813	3584	4-15-02	O
Lamzan	8192	5-18-03	O	Lanky.3153	12288	5-1-02	O
LazyMin.31	96768	4-9-04	X	Legacy	19968	3-1-04	O

TABLE C.11: Win32 Virus Detection Result by *VirusDetector* (Cont.),
Win32 Virus Total Number: 499
Error number: 103, False negative: 0.2064 (approx. 21%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is Byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Levi.2961	12288	5-16-01	O	Levi.3040	7188	11-22-03	O
Levi.3090	12288	11-22-03	O	Levi.3137	35941	11-22-03	O
Levi.3205	12288	3-1-04	X	Levi.3240	16384	8-17-02	O
Levi.3244	16384	11-22-03	X	Levi.3432	16384	11-22-03	O
Lom	341504	11-22-03	O	Lykov.a	9338	6-9-03	X
Magic.1590	8264	3-1-04	O	Magic.1922	8192	9-2-02	O
Magic.3038	12288	11-22-03	X	Magic.3078	12288	8-17-02	O
Magic.3082	8192	3-1-04	O	Mark.919	2048	6-30-03	O
Matrix.750	4096	11-22-03	O	Matrix.844	4096	4-18-02	O
Matrix.LS.1820	8192	8-26-01	O	Matrix.LS.1885	8192	11-22-03	O
Matrix.Zelda.a	4096	3-10-04	O	Matrix.Zelda.b	8192	11-22-03	O
Matrix.Zelda.c	8192	3-16-01	O	Matyas.644	45700	11-22-03	X
Maya.4106	4188	12-7-99	X	Maya.4108	8192	11-22-03	X
Maya.4113	12800	3-1-04	X	Maya.4114	8192	11-22-03	O
Maya.4161	8192	3-1-04	O	Maya.4206	8192	10-4-02	O
Maya.4254	8192	3-1-04	O	Maya.4608	8192	10-4-04	X
Melder	46080	6-27-03	O	Minit.b	10752	4-27-04	X
MircNew	25088	10-5-02	O	Mix.1852	4096	5-30-00	O
Mockoder.1120	4192	8-31-02	O	Mogul.6800	12288	3-1-04	X
Mogul.6806	12288	3-25-01	O	Mogul.6845	57344	11-22-03	O
Mogul.7189	12288	3-25-01	X	Mooder.a	8192	4-3-02	O
Mooder.d	8192	4-6-02	O	Mooder.f	14452	8-19-03	X
Mooder.g	8192	4-7-03	O	Mooder.i	8192	5-1-03	O
Mooder.j	8192	5-1-03	O	Morgoth.2560	2560	11-22-03	O
Mystery.2560	130544	12-8-01	O	NDie.2168	182504	11-22-03	O
NDie.2343	182725	11-22-03	O	Neoval	14335	5-18-03	O
NGVCK.gen	3584	10-5-04	O	Nicolam	57344	4-27-03	O
Niko.5178	65611	11-22-03	X	Noise.410	57344	3-10-02	O
Opdoc.1204	123448	6-28-03	O	Opdoc.1248	9440	6-24-03	X
Oporto.3076	37950	10-4-04	O	Padic	8192	3-1-04	O
Paradise.2116	8192	9-29-02	O	Paradise.2168	8192	9-22-02	O
Parvo	80093	3-1-04	O	Peana	8192	3-1-04	O
Perrun.a	11780	3-10-04	O	Perrun.b	5636	7-11-02	O
PGPME	86016	3-1-04	X	Pilsen.4096	4096	3-1-04	O
Positon.4668	8192	7-15-02	X	Qozah.1386	4096	1-21-99	O
Qozah.3361	8192	12-4-01	O	Qozah.3365	8192	3-1-04	O
Qozah.3370	8192	8-2-99	X	RainSong.3874	8192	11-22-03	O
RainSong.3891	61509	3-1-04	O	RainSong.3910	8192	12-5-01	X
RainSong.3956	12288	10-4-04	X	RainSong.4198	8192	3-12-02	O
RainSong.4266	12288	10-4-04	X	Ramdile	18801	3-1-04	X
Razenya	8192	10-25-03	O	Redart.2796	380972	8-172000	X
Redemption.a	16384	3-10-04	O	Redemption.b	16384	3-1-04	O
Redemption.c	7171	6-15-98	O	Refer.2939	36352	3-1-04	O
RemEx	224256	3-1-04	X	Revaz	8192	3-1-04	O

TABLE C.12: Win32 Virus Detection Result by *VirusDetector* (Cont.),
Win32 Virus Total Number: 499
Error number: 103, False negative: 0.2064 (approx. 21%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is Byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
Rever	32768	3-1-04	X	Rhapsody.2602	221	10-4-04	O
Rhapsody.2619	8192	3-1-04	O	Riccy.a	32768	3-10-04	O
Riccy.b	172032	10-28-01	O	Riccy.c	24576	11-22-03	O
Rigel.6468	106496	11-22-03	X	Rikenar.1480	8192	9-9-98	O
Rivanon	3584	6-26-03	O	Rufoll.1432	2560	2-11-02	O
Rutern.5244	9340	11-5-03	O	Ryex	8192	3-1-04	O
Sadon.900	8192	3-1-04	O	Sandman.4096	4096	7-4-02	O
Sankei.1062	8192	8-28-03	O	Sankei.1409	8192	6-8-03	O
Sankei.1455	8192	8-28-03	O	Sankei.1493	8192	6-8-03	O
Sankei.1766	8192	6-8-03	O	Sankei.1983	8192	6-8-03	O
Sankei.3001	8192	6-8-03	O	Sankei.3077	8192	6-8-03	O
Sankei.3480	8192	6-8-03	O	Sankei.3514	8192	8-6-03	O
Sankei.3580	8192	6-8-03	X	Sankei.3586	8192	6-8-03	O
Sankei.3621	8192	8-6-03	O	Sankei.4085	8192	2-8-04	O
Santana.1104	81920	12-4-01	O	Savior.1680	8192	1-8-01	O
Savior.1696	12288	5-18-01	O	Savior.1740	12288	3-28-02	O
Savior.1828	20480	8-6-01	O	Savior.1832	12288	3-1-04	O
Savior.1904	12288	12-4-01	O	Saynob.2406	5120	5-15-03	O
Segax.1136	8192	3-1-04	O	Segax.1137	8192	3-3-03	O
Segax.1160	8192	7-12-01	O	Sentinel.a	16384	3-10-04	X
Senummy.1838	8192	10-2-03	O	Seppuku.1606	8192	9-2-02	O
Seppuku.2763	30208	5-8-02	O	Seppuku.2764	30208	3-1-04	O
Seppuku.4827	102400	9-22-02	O	Seppuku.6834	12288	7-12-02	O
Seppuku.6972	12288	9-1-01	O	Seppuku.6973	12288	7-12-02	O
Seppuku.9728	12288	12-4-01	O	Shan.1842	4096	4-27-02	O
Shown.538	4096	4-6-02	O	Shown.539.a	4096	3-10-04	O
Shown.540.b	4096	5-19-03	O	Silcer	17920	3-1-04	X
Slaman.a	24576	6-28-03	O	Slaman.i	24576	7-13-04	O
Small.1144	2560	1-4-01	O	Small.1368	53248	5-5-02	O
Small.1388	8192	9-2-02	O	Small.139	94208	7-17-02	O
Small.1393	8192	1-2-02	O	Small.1416	16699	8-7-02	O
Small.1424	8192	1-2-02	O	Small.1468	8192	4-6-03	O
Small.1700	4286	8-5-02	O	Small.2218	96526	9-2-02	X
Small.2280	96588	12-31-00	X	Small.2560	8192	4-25-03	O
Smog.b	12288	10-2-03	O	Spelac.1008	4096	11-17-02	O
Spreder	595924	5-9-03	X	Staro.1538	8192	3-1-04	O
Stepar.b	39936	5-2-03	X	Stepar.dr	19456	1-1-04	X
Stepar.e	65536	5-5-99	X	Stepar.f	150528	8-23-01	X
Stepar.g	137216	8-23-01	O	Stepar.j	139264	8-23-01	O
Sugin	147456	9-19-02	O	Suns.3912	20468	9-2-02	O

TABLE C.13: Win32 Virus Detection Result by *VirusDetector* (Cont.),
Win32 Virus Total Number: 499
Error number: 103, False negative: 0.2064 (approx. 21%)
Date indicates the virus's detected & caught date in form MM-DD-YY.
Virus names are also the names of the test files. Size unit is Byte.

Virus Name	Size	Date	Detection	Virus Name	Size	Date	Detection
SWOG.based	4096	6-13-02	O	Taek.1275	6144	7-27-02	O
Tapan.3882	12288	12-31-01	O	Team.a	4096	3-10-04	O
Team.b	4096	8-30-02	O	Team.c	4096	9-22-02	O
Team.d	4096	9-14-02	O	TeddyBear	2560	3-1-04	O
Tenta.2045	10240	5-9-02	O	Test.1334	67072	9-2-02	O
This31.16896	51202	5-8-01	O	Thorin.11932	16384	3-1-04	O
Thorin.b	16384	3-1-04	O	Thorin.c	16384	10-23-99	O
Thorin.d	16384	7-14-99	O	Thorin.e	16384	10-23-99	O
Tolone	12288	2-9-03	X	Ultratt	332	9-19-01	X
Ultratt.8152	12288	3-1-04	X	Ultratt.8167	12288	10-4-02	O
Undertaker.4887	12288	11-22-03	O	Undertaker.5036.a	12288	11-22-03	O
Usem.a	16384	7-11-02	O	Usem.b	16384	7-11-02	X
Vampiro.7018	18432	3-1-04	X	Vampiro.a	16896	3-10-04	O
VbFrm	28672	7-26-02	O	VCell.3041	8192	3-31-01	O
VCell.3468	8192	8-29-01	O	VCell.3504	8192	3-1-04	O
VChain	110592	3-1-04	O	Velost.1186	8192	9-19-02	O
Velost.1233	84394	4-9-04	O	Velost.1241	56963	4-22-04	X
Vorcan	8192	10-4-04	O	Vulcano	12288	3-1-04	O
Wabrex.a	8192	3-10-04	O	Weird.10240	9216	3-1-04	O
Weird.c	83968	10-7-00	O	Weird.d	20480	11-22-03	O
Wide.8225	16896	7-30-02	X	Wide.b	12288	8-31-02	O
Wide.c	12288	8-8-02	X	Wolf.b	4096	2-27-02	O
Wolf.c	8192	10-4-04	O	Xorala	306176	3-7-04	O
Xoro.4092	6140	5-16-02	O	Yasw.1000	4096	11-22-03	O
Yasw.924	4096	12-29-00	O	Yerg.9412	28672	5-25-02	O
Yerg.9571	16384	11-22-03	O	Younga.4434	83527	5-20-01	X
Zaka.a	2809	11-1-04	X	Zawex.3196	32768	9-22-02	X
ZHymn.a	88064	4-5-01	O	ZHymn.b	90112	8-23-01	O
ZHymn.Host	10752	3-1-04	O	ZMist	86016	3-1-04	O
ZMist.d.dr	28672	3-1-04	X	ZMist.dr	28672	10-4-04	X
Zombie	19131	3-1-04	O	Zomby.17920	17920	11-22-03	O
ZPerm.a	99840	3-10-04	O	ZPerm.a2	73728	11-13-00	O
ZPerm.b	70144	3-1-04	O	ZPerm.b2	139264	11-22-03	X

TABLE C.14: Normal Executable Program's Virus Check Result by *VirusDetector*
Normal Executable File's Total Number: 80
Error number: 24, False positive: 0.3 (approx. 30%).
If the result of detection is marked X, *VirusDetector* says this file is a virus-infected file, which means incorrect detection.

Filename	Detection	Filename	Detection
dxwebsetup.exe		divx311.exe	
awsepersonal.exe		DVD2DIVXVCD_trial.exe	
paulp_en1.exe		adrenalin2.0.1.exe	
GOMPLAYER14.exe		sdvd190.exe	
csdl13.exe		HwpViewer.exe	
SSHSecureShellClient-3.2.9.exe		DivX505Bundle.exe	
klcodec220b.exe		SwansMP24a-WCP.exe	
DivXPro511GAINBundle.exe		NATEON.exe	X
WinPcap_3_1_beta_3.exe		ducp708_type3_free.exe	
wmpcdcs8.exe		Acrobat.exe	
iTunes.exe		pccmain.exe	
sicstusc.exe		Tra.exe	
acrodist.exe		java.exe	X
PCcpfw.exe		sicstus.exe	
Trialmsg.exe	X	Ad-Aware.exe	
javaw.exe	X	PCctool.exe	
splfr.exe	X	tsc.exe	
AdobeUpdateManage.exe		jpicp132.exe	X
Photoshp.exe		spmkds.exe	
conf.exe		policytool.exe	X
spmkrs.exe	X	unregaaw.exe	
CSDL.exe		kinit.exe	X
Powerpnt.exe		ssh2.exe	
Unwise.exe		csdlvw.exe	
klist.exe	X	Quicktimeplayer.exe	
Sshclient.exe		Wavtoasf.exe	
dialog_patch.exe	X	ktab.exe	X
ssh-keygen2.exe		Winword.exe	X
Directcd.exe		moviemk.exe	X
rmiregistry.exe	X	Tmntsrv.exe	
Winzip32.exe		Dreamweaver.exe	
msmsgs.exe	X	Scandisc.exe	
Tmoagent.exe		Excel.exe	
MSohtmed.exe	X	tmproxy.exe	X
iedw.exe		orbd.exe	
servertool.exe	X	tmupdito.exe	
iexplore.exe	X	PCClient.exe	
sftp2.exe		tnameserv.exe	X
uninstall.exe	X	keytool.exe	X
rmid.exe	X	scp2.exe	

Bibliography

[A. Begel and Graham(1999)] A. Begel, S. M., Graham, S. L., August 1999. Bpf+: Exploiting global data-flow optimization in a generalized packet filter architecture. Proc. of ACM SIGCOMM, Cambridge, MA, USA, pp. 123–134.

[Administrators(2003)] Administrators, 2003. Email ube statistics.
URL http://www.cpsc.ucalgary.ca/Help/internet/email/ube_ststs.php

[Ahnlab(2002)] Ahnlab, August 2002. Virus information.
URL http://home.ahnlab.com

[Anderson(2001a)] Anderson, J., March 2001a. An analysis of fragmentation attacks.
URL http://www.ouah.org/fragma.html

[Anderson(2001b)] Anderson, R. J., 2001b. Security Engineering / A Guide to Building Dependable Distributed Systems. John Wiley & Sons.

[Arnold and Tesauro(2000)] Arnold, W., Tesauro, G., 2000. Automatically generated win32 heuristic virus detection. Proceedings of the 2000 International Virus Bulletin Conference.
URL http://vx.netlux.org/lib/files/awa01/awa01.pdf

[Atkinson(1995)] Atkinson, R., August 1995. Security architecture for the internet protocol.
URL http://www.ietf.org/rfc/rfc1825.txt

[Attar and Hazelhurst(2002)] Attar, A., Hazelhurst, S., 2002. Fast packet filtering using n-ary decision diagrams.

[Authentium(2004)] Authentium, 2004. Antivirus and security threat alerts, threat name: Dumaru/w32.dumaru@mm.
URL http://www.authentium.com/threats/analysis/VirusDetail.asp?RefNo=646

[Baboescu and Varghese(2001)] Baboescu, F., Varghese, G., August 2001. Scalable packet classification. Proc. of ACM SIGCOMM, San Diego, CA, USA, pp. 199–210.

[Ballew(1997)] Ballew, S. M., 1997. Managing IP Networks with CISCO Routers. O'Reilly & Associates.

[Bellovin(1989)] Bellovin, S. M., April 1989. Security problems in the tcp/ip protocol suite. Computer Communication Review 19 (2), 32–48.
URL http://www.ja.net/CERT/Bellovin/TCP-IP_Security_Problems.html

[Benecke(1999)] Benecke, C., December 1999. A parallel packet screen for high speed networks. Proc. of the 15th Annual Computer Security Applications Conference, IEEE Computer Society, Phoenix, Arizona.
URL http://www.acsac.org/1999/papers/wed-a-1330-benecke.pdf

[Bernstein(1997)] Bernstein, D. J., 1997. Tcp/ip syn cookies.
URL http://cr.yp.to/syncookies.html

[Boyer and Moore(1977)] Boyer, R. S., Moore, J. S., October 1977. A fast string searching algorithm. Communications of the ACM 20 (10), 762–772.

[Braden(1995)] Braden, R., July 1995. T/tcp – tcp extensions for transactions, functional specification.
URL ftp://ftp.rfc-editor.org/in-notes/rfc1644.txt

[Bryant(1986)] Bryant, R. E., 1986. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers C-35(8), 677–691.
URL http://www.cs.cmu.edu/~bryant/pubdir/ieeetc86.pdf

[Bryant(1992)] Bryant, R. E., Sep 1992. Symbolic boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys 23 (3), 293–318.
URL http://www.cs.cmu.edu/~bryant/pubdir/acmcs92.pdf

[Cannady and Mahaffey(1998)] Cannady, J., Mahaffey, J., 1998. The application of artificial neural networks to misuse detection: initial results. the 1st International Workshop on Recent Advances in Intrusion Detection (RAID 1998).

[CCITT(1992)] CCITT, I.-T., 1992. Recommendation z.100: Specification and description language (sdl)General Secretariat, Geneve, Switzerland.

[CERT(1996)] CERT, 1996. Cert advisory ca-1996-26 denial-of-service attack via ping.
URL http://www.cert.org/advisories/CA-1996-26.html

[CERT(1998)] CERT, 1998. Cert advisory ca-1998-01 smurf ip denial-of-service attacks.
URL http://www.cert.org/advisories/CA-1998-01.html

[CERT(1999a)] CERT, 1999a. Cert advisory/ca-1999-04 melissa macro virus.
URL http://www.cert.org/advisories/CA-1999-04.html

[CERT(1999b)] CERT, 1999b. Cert/cc incident note in-99-03, cih/chernobyl virus.
URL http://www.cert.org/incident_notes/IN-99-03.html

[CERT(2000a)] CERT, 2000a. Cert advisory ca-1996-21 tcp syn flooding and ip spoofing attacks.
URL http://www.cert.org/advisories/CA-1996-21.html

[CERT(2000b)] CERT, May 2000b. Cert advisory ca-2000-04: Love letter worm.
URL http://www.cert.org/advisories/CA-2000-04.html

[CERT(2001a)] CERT, July 2001a. Cert advisory ca-2001-22: W32/sircam malicious code.
URL http://www.cert.org/advisories/CA-2001-22.html

[CERT(2001b)] CERT, September 2001b. Cert advisory ca-2001-26: Nimda worm.
URL http://www.cert.org/advisories/CA-2001-26.html

[CERT(2001c)] CERT, November 2001c. Cert incident note in-2001-14: W32/badtrans worm.
URL http://www.cert.org/incident_notes/IN-2001-14.html

[CERT(2002a)] CERT, 2002a. Cert advisory ca-2001-23: Continued threat of the code red worm.
URL http://www.cert.org/advisories/CA-2001-23.html

[CERT(2002b)] CERT, January 2002b. Cert incident note in-2002-01: W32/myparty malicious code.
URL http://www.cert.org/incident_notes/IN-2002-01.html

[CERT(2002c)] CERT, March 2002c. Cert incident note in-2002-02: W32/gibe malicious code.
URL http://www.cert.org/incident_notes/IN-2002-02.html

[Chantico(1992)] Chantico, 1992. Combating Computer Crime: Prevention, Detection, Investigation. McGraw-Hill, Inc, chantico Publishing Company, Inc.

[Cheswick and Bellovin(1994)] Cheswick, W. R., Bellovin, S. M., 1994. Firewalls and Internet Security: Repelling the Wily Hacker. Addison Wesley.

[Cheung and McCanne(1999)] Cheung, G., McCanne, S., 1999. Dynamic memory model based framework for optimization of ip address lookup algorithms. Proc. of the 7th Annual International Conference on Network Protocols, Toronto, Canada, pp. 11–20.

[Chmielarski(2001)] Chmielarski, T., 2001. Reconnaissance techniques using spoofed ip addresses, sans intrusioin detection faq.
URL http://www.sans.org/resources/idfaq/spoofed_ip.php

[Cinderella(2003)] Cinderella, 2003. Cinderella sdl.
URL http://www.cinderella.dk

[CISCO(1997)] CISCO, 1997. Security advisory: Tcp loopback dos attack (land.c) and cisco devices.
URL http://www.cisco.com/warp/public/770/land-pub.shtml

[CISCO(2002)] CISCO, 2002. Ciscoworks access control list manager 1.4 overview.
URL http://www.cisco.com/warp/public/cc/pd/wr2k/caclm/prodlit/aclm_ov.htm

[Crocker(1982)] Crocker, D. H., 1982. Standard for the format of arpa-internet text messages, rfc 822.
URL http://www.ietf.org/rfc/rfc0822.txt

[D. Decasper and Plattner(1998)] D. Decasper, Y. Dittia, G. P., Plattner, B., October 1998. Router plugins: A software architecture for next generation routers. Computer Communication Review 28 (4), 229–240.

[D. Michie and (Eds)(1994)] D. Michie, D. J. S., (Eds), C. C. T., February 1994. Machine Learning, Neural and Statistical Classification.
URL http://www.amsta.leeds.ac.uk/~charles/statlog/

[Davis and Weyuker(1983)] Davis, M. E., Weyuker, E. J., 1983. Computability, Complexity, and Languages. Academic Press.

[Dittrich(1999a)] Dittrich, D., October 1999a. The dos project's trinoo distributed denial of service attack tool.
URL http://staff.washington.edu/dittrich/misc/trinoo.analysis

[Dittrich(1999b)] Dittrich, D., December 1999b. The stacheldraht distributed denial of service attack tool.
URL http://staff.washington.edu/dittrich/misc/stacheldraht.analysis

[Dittrich(1999c)] Dittrich, D., October 1999c. The tribe flood network distributed denial of service attack tool.
URL http://staff.washington.edu/dittrich/misc/tfn.analysis

[eEye(2001)] eEye, 2001. .ida code red worm.
URL http://www.eeye.com/html/research/advisories/2l20010717.html

[Eichin and Rochlis(1988)] Eichin, M., Rochlis, J., 1988. With microscope and tweezers: An analysis of the internet virus of november 1988.

[Eichin and Rochlis(1989)] Eichin, M., Rochlis, J., 1989. With microscope and tweezers: An analysis of the internet virus of november 1988. IEEE Computer Society Symposium on Security and Privacy.

[Ellermann and Benecke(1998)] Ellermann, U., Benecke, C., June 1998. Firewalls for atm networks. Proc. of INFOSEC'COM.
URL http://www.cert.dfn.de/eng/team/ue/fw/fire-atm

[Esa Alhoniemi and Vesanto(2002)] Esa Alhoniemi, Johan Himberg, J. P., Vesanto, J., 2002. Som toolbox 2.0, a software library for matlabSOM Toolbox team, Laboratory of Computer and Information Science, Finland.
URL http://www.cis.hut.fi/projects/somtoolbox/

[F. Baker and Smith(2002)] F. Baker, K. C., Smith, A., May 2002. Management information base for the differentiated services architecture, darpa internet program protocol specification, rfc3289.
URL http://www.ietf.org/rfc/rfc3289.txt

[Feldmann and Muthukrishnan(2000)] Feldmann, A., Muthukrishnan, S., March 2000. Tradeoffs for packet classification. Proc. of IEEE INFOCOMM, Tel-Aviv, Israel, pp. 1193–1202.

[Ferguson and Senie(2000)] Ferguson, P., Senie, D., May 2000. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing, rfc2827.
URL ftp://ftp.rfc-editor.org/in-notes/rfc2827.txt

[Ferrie(2002)] Ferrie, P., 2002. W32/klez.
URL http://toronto.virusbtn.com/magazine/archives/200207/klez.xml

[Fisher(1925)] Fisher, R. A., 1925. What has now appearedStatistical Methods for Research Workers, Edinburgh: Oliver and Boyd.

[Fyodor(1997)] Fyodor, 1997. The land attack(ip dos).
URL http://www.insecure.org/sploits/land.ip.DOS.html

[Fyodor(2003)] Fyodor, 2003. Idle scanning and related ipid games.
URL http://www.insecure.org/nmap/idlescan.html

[G. Tesauro and Sorkin(1996)] G. Tesauro, J. O. K., Sorkin, G. B., August 1996. Neural networks for computer virus recognition. IEEE Expert 11 (4), 5–6.

[G. Ziemba and Traina(1995)] G. Ziemba, D. R., Traina, P., October 1995. Security considerations for ip fragment filtering, rfc 1858.
URL ftp://ftp.rfc-editor.org/in-notes/rfc1858.txt

[Ginossar and Trope(1987)] Ginossar, Z., Trope, Y., 1987. Problem solving in judgment under uncertainty. Journal of Personality and Social Psychology 52, 464–473.

[Grossman(2002)] Grossman, D., 2002. New terminology and clarifications for diffserv, rfc3260, network working group.
URL ftp://ftp.rfc-editor.org/in-notes/rfc3260.txt

[Gryaznov(1999)] Gryaznov, D., 1999. Scanners of the year 2000: Heuristics. Proceedings of the 5th International Virus Bulletin.
URL http://vx.netlux.org/texts/html/scan2000.html

[Gupta and Mukherjee(1996)] Gupta, B., Mukherjee, B., March 1996. Network security via reverse engineering of tcp code: Vulnerability analysis and proposed solutions. Proc. of IEEE Infocom'96, San Francisco, CA, USA, pp. 603–610.

[Gupta and McKeown(1999a)] Gupta, P., McKeown, N., August 1999a. Packet classificaiton on multiple fields. Proc. of ACM SIGCOMM, Cambridge, MA, USA, pp. 147–160.

[Gupta and McKeown(1999b)] Gupta, P., McKeown, N., 1999b. Packet classificaiton using hierarchical intelligent cuttings. Proc. of Hot Interconnects VII.

[Haykin(1999)] Haykin, S., 1999. Neural Networks: A Comprehensive Foundation, International Edition/Second Edition. Prentice Hall.

[Hazelhurst(1999)] Hazelhurst, S., 1999. Algorithms for analysing firewall and router access lists.

[Heffernan(1998)] Heffernan, A., 1998. Protection of bgp sessions via the tcp md5 signature option, rfc2385.
URL `http://www.ietf.org/rfc/rfc2385.txt?number=2385`

[Hinton and Sejnowski(1999)] Hinton, G., Sejnowski, T. J., June 1999. Unsupervised Learning: Foundations of Neural Computation. The MIT Press.

[Hoggan(2000)] Hoggan, D., 2000. The internet book: Introduction and reerence.
URL `http://www.camtp.uni-mb.si/books/Internet-Book/IP_TeardropAttack.html`

[Hopcroft and Ullman(1979)] Hopcroft, J. E., Ullman, J. D., 1979. Introduction to Automata Theory, languages, and computation. Addison Wesley.

[Hundley and Held(2000)] Hundley, K., Held, G., March 2000. Cisco Access lists Field Guide. McGraw-Hill.

[InSeon and Ultes-Nitsche(2002)] InSeon, Ultes-Nitsche, U., July 2002. An integrated network security approach : Pairing detecting malicious patterns with anomaly detection. Proc. Conference on Korean Science and Engineering Association in UK (KSEAUK2002).

[J. Mogul and Accetta(1987)] J. Mogul, R. R., Accetta, M., November 1987. The packet filter: An efficient mechanism for user-level network code. Proc. of the 11th ACM Symposium on Operating Systems Principles, ACM Press. An updated version is available as DEC WRL Research Report 87/2, pp. 39–51.

[Jan Ellsberger and Sarma(1997)] Jan Ellsberger, D. h., Sarma, A., 1997. SDL : Formal Object-oriented Language for Communicating Systems. Prentice Hall.

[John T. McHenry and Cocks(1997)] John T. McHenry, Patrick W. Dowd, T. M. C. F. A. P., Cocks, W. B., 1997. An fpga-based coprocessor for atm firewalls. Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines, pp. 30–39.

[Julia Allen(2000)] Julia Allen, Alan Christie, W. F. J. M. J. P. E. S., 2000. State of the practice of intrusion detection technologiesTechnical Report, CMU/SEI-99-TR-028, Carnegie Mellon University, Software Engineering Institute.

[K. Nichols and Black(1998)] K. Nichols, S. Blake, F. B., Black, D., Dec 1998. Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers, rfc 2474.

[KASPERSKY(1994-2005)] KASPERSKY, 1994-2005. Windows viruses.
URL `http://www.avp.ch/avpve/newexe.stm`

[KASPERSKY(2003)] KASPERSKY, 2003. Win32.apparition.
URL `http://www.avp.ch/avpve/newexe/win32/appar32.stm`

[Kaspersky(2000)] Kaspersky, E., 2000. Virus analysis texts - macro viruses.
URL `http://www.avp.ch/avpve/classes/macrovir.stm`

[Kephart(1994)] Kephart, J. O., 1994. A biologically inspired immune system for computers. Artificial Life IV, pp. 130–193, proceedings of the Fourth International Workshop on Synthesis and Simulation of Living Systems, Rodney A. Books and Pattie Maes, eds.
URL `http://www.research.ibm.com/antivirus/SciPapers/Kephart/ALIFE4/alife4.distrib.h`

[Kohonen(1982)] Kohonen, T., 1982. Self-organized formation of topologically correct feature maps. Biological Cybernetics 43, 59–69.

[Kohonen(1988)] Kohonen, T., 1988. Self-Organization and Associative Memory. New York: Springer-Verlag, 3rd ed.

[Kohonen(1995)] Kohonen, T., 1995. Self-Organizing Maps. Springer, Berlin, Heidelberg.

[Kohonen(1999)] Kohonen, T., 1999. Comparison of som point densities based on different criteria. Neural Computation 11 (8), 2081–2095.

[Lahey(2000)] Lahey, K., 2000. Tcp problems with path mtu discovery, rfc2923.
URL `ftp://ftp.rfc-editor.org/in-notes/rfc2923.txt`

[Lakshman and Stiliadis(1998)] Lakshman, T. V., Stiliadis, D., September 1998. High speed policy-based packet forwarding using efficient multidimensional range matching. Proc. of ACM SIGCOMM, Vancouver, Canada, pp. 203–214.

[Langley and Sage(1994)] Langley, P., Sage, S., 1994. Induction of selective bayesian classifiers. Proceedings of the 10th Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann.
URL `http://www.isle.org/~langley/papers/select.uai94.ps.gz`

[Lanning(1987)] Lanning, K., 1987. Some reasons for distinguishing between nonnormative response and irrational decision. Journal of Psychology 12, 109–117.

[Lear(2000)] Lear, A. C., June 2000. New chip helps with network security. IEEE Computer 33 (6), 24.

[Lee and Heinbuch(2001)] Lee, S. C., Heinbuch, D. V., July 2001. Training a neural network-based intrusion detector to recognize novel attacks. IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans 31 (4), 294–299.

[Leech(2003)] Leech, M., 2003. Key management considerations for the tcp md5 signature option, rfc3562.
URL `ftp://ftp.rfc-editor.org/in-notes/rfc3562.txt`

[Linn(1988)] Linn, J., 1988. Privacy enhancement for internet electronic mail: Part i: Message encipherment and authentication procedures, rfc 1040.
URL `http://www.ietf.org/rfc/rfc1040.txt`

[M. Handley and Paxson(2001)] M. Handley, C. K., Paxson, V., 2001. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. Proc. of the 10th USENIX Security Symposium (Security '01).

[M. Handley and Floyd(2000)] M. Handley, J. P., Floyd, S., 2000. Tcp congestion window validation, rfc2861.
URL `http://www.ietf.org/rfc/rfc2861.txt?number=2861`

[M. L. Bailey and Sarkar(1994)] M. L. Bailey, B. Gopal, M. A. P. L. L. P., Sarkar, P., Nov 1994. Pathfinder: A pattern-based packet classifier. Proc. of the 1st Symposium on Operating System Design and Implementation, USENIX Association, pp. 115–123.

[M. Mathis and Romanow(1996)] M. Mathis, J. Mahdavi, S. F., Romanow, A., 1996. Tcp selective acknowledgment options, rfc 2018.
URL `http://www.ietf.org/rfc/rfc2018.txt`

[M. Yuhara and Moss(1994)] M. Yuhara, B. N. Bershad, C. M., Moss, J. E. B., January 1994. Efficient packet demultiplexing for multiple endpoints and large messages. Proc. of the 1994 Winter USENIX Conference, pp. 153–165.

[Marcel Waldvogel and Plattner(1997)] Marcel Waldvogel, George Varghese, J. T., Plattner, B., 1997. Scalable high speed ip routing lookups. SIGCOMM, pp. 25–36.

[MATHWORKS(2003)] MATHWORKS, 2003. The mathworks, inc.MATLAB.
URL `http://www.mathworks.com`

[Matthew G. Schultz and Zadok(2001a)] Matthew G. Schultz, E. E., Zadok, E., May 2001a. Data mining methods for detection of new malicious executables. IEEE Symposium on Security and Privacy (IEEE S & P 2001).

[Matthew G. Schultz and Zadok(2001b)] Matthew G. Schultz, E. E., Zadok, E., June 2001b. Mef: Malicious email filter, a unix mail filter that detects malicious windows executables. USENIX Annual Technical Conference - FREENIX Track.
URL `http://www.cs.columbia.edu/ids/publications/mef-freenix01.pdf`

[McAfee(2001)] McAfee, 2001. Virus name: W32/semisoft.58368d, mcafee virus characteristic.
URL `http://vil.nai.com/vil/content/v_99264.htm`

[McAfee(2002)] McAfee, 2002. Mcafee home page.
URL `http://www.mcafee.com`

[McCanne and Jacobson(1993)] McCanne, S., Jacobson, V., 1993. The bsd packet filter: A new architecture for user-level packet capture. Proc. of USENIX Winter Conference, pp. 259–269.

[MessageLabs(2002)] MessageLabs, 2002. W32/bugbear-ww, message labs.
URL `http://www.messagelabs.com/viruseye/report.asp?ip=110`

[Nazario(2004)] Nazario, J., 2004. Defense and Detection Strategies against Internet Worms. ARTECH HOUSE, computer Security Series.

[Neale(1999)] Neale, R., Feb 1999. Is content addressable memory the key to network success? Electronic Engineering 71 (865), 9–12.

[Newman(1999)] Newman, D., January 1999. Firewall on a chip: Fore's fsa boosts throughput to multigigabit rates. Data Communications 28 (1), 44–45.

[NSFISSI(1997)] NSFISSI, August 1997. National information systems security (infosec) glossary, nsfissi no.4009.

[Oetiker and Rand(2003)] Oetiker, T., Rand, D., 2003. Multi router traffic grapher.
URL `http://people.ee.ethz.ch/~oetiker/webtools/mrtg/`

[O.Kephart and C.Arnold(1994)] O.Kephart, J., C.Arnold, W., 1994. Automatic extraction of computer virus signatures. 4th Virus Bulletin International Conference, pp. 178–184.
URL `http://www.research.ibm.com/antivirus/SciPapers/Kephart/VB94/vb94.html`

[Oppliger(1998)] Oppliger, R., 1998. Internet and Intranet Security. Artech House Inc., Boston.

[Pearl(1988)] Pearl, J., 1988. Probabilistic Reasoning In Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann Publishers, Inc., San Mateo, California.

[Pfleeger(1997)] Pfleeger, C. P., 1997. Security in Computing. Prentice-Hall International, Inc., international Edition, Second Edition.

[Postel(1980)] Postel, J., 1980. User datagram protocol, rfc 768.
URL http://www.ietf.org/rfc/rfc0768.txt

[Postel(1981a)] Postel, J., 1981a. Internet control message protocol (icmp), rfc 792, darpa internet program protocol specification.
URL http://www.ietf.org/rfc/rfc0792.txt

[Postel(1981b)] Postel, J., September 1981b. Internet protocol, rfc 791, darpa internet program protocol specification, defense advanced research projects, information sciences institute.
URL http://www.ietf.org/rfc/rfc791.txt

[Postel(1981c)] Postel, J., September 1981c. Transmission control protocol, rfc 793, darpa internet program protocol specification, defense advanced research projects, information sciences institute.
URL http://www.ietf.org/rfc/rfc793.txt

[Postel(1982)] Postel, J. B., August 1982. Simple mail transfer protocol, rfc0821.
URL http://www.ietf.org/rfc/rfc0821.txt

[Postini(2004)] Postini, 2004. Postini email stat track.
URL http://www.postini.com/stats

[R. Lo and Olsson(1995)] R. Lo, K. L., Olsson, R., 1995. Mcf: a malicious code filter. Computers & Security 14 (6), 541–566.
URL http://seclab.cs.ucdavis.edu/papers/llo95.ps

[Ramakrishnan and Floyd(1999)] Ramakrishnan, K., Floyd, S., Jan 1999. A proposal to add explicit congestion notification (ecn) to ip, rfc 2481, network wroking group.
URL http://www.ietf.org/rfc/rfc2481.txt

[R.L.Ziegler(2000)] R.L.Ziegler, 2000. Linux Firewalls. New Riders, Indianapolis, Indiana.

[Rubin and Geer(1998)] Rubin, A., Geer, D., 1998. Mobile code security. IEEE Internet Computing 2 (6), november/December.

[S. Hazelhurst and Sinnappan(2000)] S. Hazelhurst, A. A., Sinnappan, R., June 2000. Algorithms for improving the dependability of firewall and filter rule lists. Proc. of the International Conference on Dependable Systems and Networks, pp. 576–585.

[Salcic and Smailagic(1997)] Salcic, Z., Smailagic, A., 1997. Digital Systems Design and Prototyping Using Field Programmable Logic. Kluwer Academic, Boston.

[Samamura(1998)] Samamura, M., 1998. W95.cih, volume expanded threat list and virus encyclopaedia.
URL http://securityresponse.symantec.com/avcenter/venc/data/cih.html

[Seeley(1988)] Seeley, D., 1988. A tour of the worm.

[Service(1996)] Service, E. N., 1996. Windows 95 virus-boza announced.
URL http://www.emergency.com/boza.htm

[Shankar and Paxson(2003)] Shankar, U., Paxson, V., 2003. Active mapping: Resisting nids evasion without altering traffic. Proc. of the IEEE Symposium on Security and Privacy '03).

[Sophos(2002)] Sophos, 2002. anti-virus product.
URL http://www.sophos.co.uk

[Sophos(2005)] Sophos, September 2005. Top ten viruses and hoaxes reported to sophos in september 2005.
URL http://www.sophos.com/pressoffice/pressrel/uk/toptensep05.html

[Spafford(1988)] Spafford, E. H., 1988. The internet worm program: An analysisPurdue Technical report CSD-TR-823.

[Stevens(1994)] Stevens, W. R., 1994. TCP/IP Illustrated Vol. 1 - The Protocols. Addison-Wesley.

[Stevens and Wright(1995)] Stevens, W. R., Wright, G. R., 1995. TCP/IP Illustrated Vol.2 - The Implementation. Addison-Wesley.

[Swimmer(2000)] Swimmer, M., 2000. Review and outlook of the detection of viruses using intrusion detection systems. the 3rd International Workshop on Recent Advances in Intrusion Detection (RAID 2000).

[Symantec(2002)] Symantec, 2002. Symantec worldwide homepage.
URL http://www.symantec.com/product/

[TrendMicro(2002a)] TrendMicro, 2002a. Housecall, trend micro - free online virus scan.
URL http://housecall.trendmicro.com

[TrendMicro(2002b)] TrendMicro, 2002b. Trend micro virus protection products.
URL http://www.trendmicro.com/en/products/global/enterprise.htm

[Ultes-Nitsche and Yoo(2003)] Ultes-Nitsche, U., Yoo, I., July 2003. Steps toward and intelligent firewall - a basic model. Proc. Conference on Information Security for South Africa (ISSA2003).

[Ultes-Nitsche and Yoo(2004)] Ultes-Nitsche, U., Yoo, I., June/July 2004. Run-time protocol-conformance verification in firewalls. Proc. of the 4th Annual ISSA 2004 IT Security Conference.

[V. B. Hinsz and Robertson(1988)] V. B. Hinsz, R. S. Tindale, D. H. N. J. H. D., Robertson, B. A., 1988. The influence of the accuracy of individuating information

on the use of base rate information in probability judgment. Journal of Experimental Social Psychology 24, 127–145.

[V. Gill and Meyer(2004)] V. Gill, J. H., Meyer, D., 2004. The generalized ttl security mechanism (gtsm), rfc 3682.
URL ftp://ftp.rfc-editor.org/in-notes/rfc3682.txt

[V. Jacobson and Borman(1992)] V. Jacobson, R. B., Borman, D., 1992. Tcp extensions for high performance, rfc1323.
URL ftp://ftp.rfc-editor.org/in-notes/rfc1323.txt

[V. Paxson and Volz(1999)] V. Paxson, M. Allman, S. D. W. F. J. G. I. H. K. L. J. S., Volz, B., 1999. Known tcp implementation problems, rfc 2525.
URL http://www.ietf.org/rfc/rfc2525.txt

[V. Srinivasan and Varghese(1999)] V. Srinivasan, S. S., Varghese, G., 1999. Packet classification using tuple space search. SIGCOMM, pp. 135–146.

[Venkatachary Srinivasan and Waldvogel(1998)] Venkatachary Srinivasan, George Varghese, S. S., Waldvogel, M., 1998. Fast and scalable layer four switching. SIGCOMM, pp. 191–202.

[VirusBulletin(2004)] VirusBulletin, 2004. Viruses: Vbs/bubbleboyFirst active date: 8 November 1999.
URL http://www.virusbtn.com/resources/viruses/indepth/bubbleboy.xml

[Vos and Konijnenberg(1996)] Vos, J., Konijnenberg, W., Nov 1996. Linux firewall facilities for kernel-level packet screening.
URL http://www.xos.nl/linux/ipfwadm/paper

[Wang and Liu(1993)] Wang, C. J., Liu, M. T., 1993. Generating test cases for efsm with given fault models. Proc. of IEEE INFOCOM93, Vol.2, pp.774-781.

[Wang(1998)] Wang, R., 1998. Flash in the pan?

[White(1998)] White, S. R., 1998. Open problems in computer virus researchIBM online publication.
URL http://www.research.ibm.com/antivirus/SciPapers/White/Problems/Problems.html

[Wikipedia(2005)] Wikipedia, 2005. Phishing.
URL http://en.wikipedia.org/wiki/Phishing

[WildList(2001)] WildList, 2001. Virus descriptions of viruses in the wild.
URL http://www.f-secure.com/virus-info/wild.html

[Xilinux(2001)] Xilinux, 2001. Asic alternatives.
URL http://www.xilinx.com/

[Yoo(2004a)] Yoo, I., October 2004a. Adaptive firewall model to detect email viruses. Proc. of the 38th Annual IEEE International Carnahan Conference on Security Technology, ICCST 2004.

[Yoo(2004b)] Yoo, I., 2004b. An Intelligent Firewall Architecture Model To Detect Internet-Scale Virus Attacks. University of Southampton, master of Philosophy (MPhil) thesis, Declarative Systems and Software Engineering Research Group, School of Electronic and Computer Science.

[Yoo(2004c)] Yoo, I., June 2004c. Protocol anomaly detection and verification. Proc. of the 5th Annual IEEE Information Assurance Workshop.

[Yoo(2004d)] Yoo, I., October 2004d. Visualizing windows executable viruses using self-organizing maps. Proc. of the 11th ACM Conference on Computer and Communications Security (CCS 2004), Workshop on Visualization and Data Mining for Computer Security (VizSEC/DMSEC-04).

[Yoo and Ultes-Nitsche(2002a)] Yoo, I., Ultes-Nitsche, U., November 2002a. Intelligent firewall: Packet-based recognition against internet-scale virus attacks. Conference on Communications and Computer Networks (CCN 2002).

[Yoo and Ultes-Nitsche(2002b)] Yoo, I., Ultes-Nitsche, U., November 2002b. An intelligent firewall to detect novel attacks an integrated approach based on anomaly detection against virus attacks. Proc. SOFSEM Conference, SOFSEM 2002 Student Research Forum, pp. 59–64.

[Yoo and Ultes-Nitsche(2003)] Yoo, I., Ultes-Nitsche, U., December 2003. Adaptive detection of worms/viruses in firewalls. Proc. of International Conference on Communication, Network, and Information Security (CNIS 2003).

[Yoo and Ultes-Nitsche(2004a)] Yoo, I., Ultes-Nitsche, U., April 2004a. How to predict email viruses under uncertainty. Proc. of the 23rd IEEE International Performance, Computing and Communications Conference, IPCCC 2004, Workshop of Information Assurance (WIA 04).

[Yoo and Ultes-Nitsche(2004b)] Yoo, I., Ultes-Nitsche, U., August 2004b. Towards runtime protocol anomaly detection and verification. Proc. of the 1st International Conference on E-Business and Telecommunication Networks, ICETE 2004.

Curriculum Vitae

C.1 InSeon Yoo

C.1.1 Education

2003.2 - 2006.5	PhD candidate, *Thesis: Justifying Anomaly Packets with Computing Methodologies & The Underlying Firewall System* Department of Computer Science, University of Fribourg, Fribourg, Switzerland.
2001.10 - 2003.1	Master of Philosophy (MPhil), Computer Science, University of Southampton, Southampton, UK. *MPhil Thesis: An Intelligent Firewall Architecture Model To Detect Internet-Scale Virus Attacks.* (MPhil Award at 10th May 2004.)
1996 - 1998	Master of Science , Computer Science, Sogang Univ, Seoul, Korea. *Thesis: A Daisy-chain scheme for delivery of personalized push-documents on WWW*
1992 - 1996	Bachelor of Science, Computer Science, Dongduk Women's Univ, Seoul, Korea.

C.1.2 Research Interests

1. Server side Computer Security and Network Security
2. Reasoning and Treating Uncertainty based on Bayesian Networks
3. Machine Learning based on Neural Networks
4. Pattern Recognition and Classification of network packets
5. Security Engineering

6. Information Warfare & Cyberwarfare - Cybercrime, Cyberterrorism
7. Human Behaviour - Human Economic Behaviour (microeconomics)

C.1.3 Research Experience

- **PhD/Research assistant, Univ of Fribourg , 2003.2 - 2005.12**
  1. Responsible for Janus Project.
  2. Organized isolated testbed subnetwork for Janus firewall.
  3. Achieved Janus & VirusDetector design/development.
  4. For justifying anomaly in packets, proposed and designed to apply computing methodologies & software engineering for examining data packets in the firewall.
  5. Achieved email classification with symbolic & algebraic manipulation (Ordered Binary Decision Diagram), and Bayesian networks for representation and inference.
  6. Achieved protocol anomaly detection & verification with protocol sanity normalization and a proposed TCP protocol verification model.
  7. Achieved non-signature based virus detection with SOM (self-organizing map).
  8. Achieved Janus firewall with the components. ASM (abstract state machine) and SDL (specification and description language) were used to design and implement the project.
- **MPhil/Research assistant, Univ of Southampton, 2001.10 - 2003.1**
  1. Researched and planned how to design an intelligent firewall.
  2. Surveyed machine learning and neural network technologies, then decided to choose and designed Bayesian network and SOM for appropriate methods to detect any abnormal packet contents.
  3. Achieved analysis of Internet viruses and Spam.
- **Staff Research/Software Engineer, ThinkFree.com, Haansoft USA & Korea, 3003 North First St, Suite 208, San Jose, CA 95134, July 99 July 2001.**
  1. Responsible for sever side relative research, design, development, and implementation.
  2. Server based development with C, Java, Servlet/JSP, and JDBC.
  3. Contributed to server team from scratch.
  4. Achieved HTTP Tunneling, security & cryptography methods inside fileserver & server systems.

5. Contributed server/network performance and maintenance with traffic monitoring & log analysing systems.
6. Contributed asynchronous file system development, achieved DB schema design, and applied the DB schema for most side development including admin systems using Informix, Oracle, mySql, hSql.
7. Proposed document management engine, and distributed resource versioning systems.
8. Organized billing systems through web sites.

- **Staff Research/Software Engineer, Simmany, DACOM (Data & Telecommunication) Corporation, LG, Korea, June 98 - June 99.**

1. Responsible for Lite Search Engine.
2. Designed lite search engine, and achieved to develop Smlite (Simmany Lite Search Engine) based on Unix for searching web document. Developed multithreaded serch engine with background real-time indexing/query responding, indexing process using gdbm, controlled it with a centre monitor program, then displayed query result with CGI.
3. Smlite was porting to several OS platforms: Sun Solarix 2.5x, NCR UNIX SVR4 MP-RAS, IBM AIX 4.2/4.3.2, DEC-UNIX 4.0, HP-10.20, SGI IRIX 64-6.2, Linux 2.1.x.
4. Achieved to develop pthread HPUX 10.20 version.
5. Planned to apply information retrieval protocol Z39.50 gateway for advanced classification and digital library.

- **MS/Research Assistant, Sogang Univ, 1996 - 1998**

1. Achieved to develop an electronic library with web CGI. 98.1 - 98.2
2. Achieved to develop a high performance multithreaded web server for e-commerce. 97.5 - 97.11 (Supported by Electronics and Telecommunications Research Institute, SERI 1997).
3. Achieved to develop a remote multimedia conference system using H.261. 96.11 - 97.2 (Supported by NAKIYUN Co.Ltd.)
4. Achieved to develop a multimedia KIOSK System with Java. 96.7 - 96.9

C.1.4 Awards & Certificates

- **Research scholarship** from Department of Electronics & Computer Sciences, University of Southampton, 2002-2003.
- **Certificate** of Accomplishment, Course Title "Oracle 8i Administrator", Oracle Korea 2000/10/23-27.

- o **Certificate** of Accomplishment, Course Title "Using the I-DataBlade API to Build DataBlade Module", Informix Korea Ltd. 1998/08/10-08/11.
- o **Scholarship** from Director, Dongduk Univ. 1989, 1991, 1992.
- o **Scholarship** from Director, Sogang Univ. 1997.

C.1.5 Extracurricular Activities

- **Leader of Cyber discussion group in PC HiTEL**. 1992.6 - 1995.1
  1. Organized Womens right discussion group
  2. Proposed various women relative topic for discussion
  3. Increased registered members including mens number
  4. Coordinated to discuss each topic with both sides view
  5. Organized regular off-line meetings for members
- **HiMEM**, Undergraduate student Membership in Korea Telecommunication (KT), HiTEL. 1995.7 - 1996.7
  1. Leader of Web Team.
  2. Introduced to students & KT about Internet new technologies.
  3. Proposed Web based cyberspace from PC database-based systems.
  4. Leading regular meeting for seminar & studying about Internet new technologies.
  5. Contributed to KT about conversion necessity & future direction for cyber space.
- **Leader of BIGS(Base Implementation Group Study) in Dongduk Univ**. 1993 - 1995
  1. Leader of initial members
  2. Proposed necessity of extra curriculum learning.
  3. Organized group for self-studying beyond school curriculum.
  4. Contributed to several computer program exhibitions through our results.
  5. Got much attention to CS department through our group from outside of the university.
- **University Student Activity Board**, 1994
  1. Responsible of study department
  2. Proposed summer computer learning session for students & neighbours

3. Organized teaching curriculum for summer computer lesson
4. Increased interests of CS department & the university from neighbours & impact of advertisement through the summer computer lesson

- Teaching Computer in Univ and Computer School, 1993 - 1998.
- Speaker in Workshop WWW-KR 5th 1997.5.
- Freelance writer in Computer Magazines : Hello PC, PC Seoul, PC Line, PC Seoul, Microsoft. 1992 - 1998.

C.1.6 Computer Technical Experiences

Computer Language	
C, Java	Multithreaded Programming, Concurrent Software Design, UNIX Programming, POSIX Programming, Network Programming.
etc	Borne Shell, Korn Shell, C Shell, BASIC, PASCAL, FORTRAN, Scheme, LISP, HTML, LaTex., Perl, Matlab. ASML, UML, SDL
Operating Systems	MSX series, MS-DOS, MS WINDOWS 3.1/95/NT, UNIX System V, IRIX64, FreeBSD, SunOS 4.X, Solaris 5.X, Linux, NCR, HPUX 9.X/10.X, DIGITAL UNIX V4.0 alpha, AIX 4.X.
Database	Informix, Oracle, mySql, hsql, pgsql, postgreSQL
CaseTools	UML CaseTool (Rational Rose), SDL tool (Cinderella), Visio

C.1.7 Publications

1. InSeon Yoo and Ulrich Ultes-Nitsche, **"Non-Signature Based Virus Detection: Towards Establishing Unknown Virus Detection Technique Using SOM"**, To appear, Journal in Computer Virology, Volume 2, Issue 3, Springer Paris, 2006.
2. InSeon Yoo, **"Visualizing Windows Executable Viruses Using Self-Organizing Maps"**, Proc. of the 11th ACM Conference on Computer and Communications Security (CCS 2004), Workshop on Visualization and Data Mining for Computer Security (VizSEC/DMSEC-04), Washington DC, USA, October 25-29, 2004.
3. InSeon Yoo, **"Adaptive Firewall Model to Detect Email Viruses"**, Proc. of the 38th Annual IEEE International Carnahan Conference on Security Technology (ICCST 2004), Albuquerque, New Mexico, USA, October 11-14, 2004.
4. InSeon Yoo and Ulrich Ultes-Nitsche, **"Towards Run-time Protocol Anomaly Detection and Verification"**, Proc. of the 1st International Conference on

E-Business and Telecommunication Networks (ICETE 2004), Setubal, Portugal, 25-28 August, 2004.

5. Ulrich Ultes-Nitsche and InSeon Yoo, **"Run-time Protocol-Conformance Verification in Firewalls"**, Proc. of the 4th Annual ISSA 2004 IT Security Conference, Johannesburg, South Africa, 30 June-2 July, 2004.

6. InSeon Yoo, **"Protocol Anomaly Detection and Verification"**, Proc. of the 5th Annual IEEE Information Assurance Workshop, West Point, New York, USA, June 10-11, 2004.

7. InSeon and Ulrich Ultes-Nitsche, **"How to Predict Email Viruses under Uncertainty"**, Prof. of IEEE Workshop on Information Assurance (WIA04) at the IEEE International Performance Computing and Communications Conference, Pheonix, Arizona, USA, 14-17 April, 2004.

8. InSeon and Ulrich Ultes-Nitsche, **"Adaptive Detection of Worms/Viruses in Firewalls"**, Proc. of International Conference on Communication, Network, and Information Security (CNIS 2003), New York, USA, 10-12 Dec, 2003.

9. Ulrich Ultes-Nitsche and InSeon Yoo, **"Steps Toward and Intelligent Firewall - A Basic Model"**, Proc. of Conference on Information Security for South Africa (ISSA2003), Sandton, Gauteng, South Africa, 9-11 July 2003.

10. InSeon Yoo and Ulrich Ultes-Nitsche, **"An Intelligent Firewall To Detect Novel Attacks : An Integrated Approach based on Anomaly Detection Against Virus Attacks."**, Proc. of SOFSEM Conference, SOFSEM 2002 Student Research Forum. Milovy, Czech Republic, Nov. 24-28, 2002.

11. InSeon Yoo and Ulrich Ultes-Nitsche, **"Intelligent Firewall: Packet-Based Recognition Against Internet-Scale Virus Attacks."**, Proc. of Conference on Communications and Computer Networks (CCN 2002), Cambridge, USA. Nov. 04-06, 2002.

12. Ulrich Ultes-Nitsche and InSeon Yoo, **"An Intelligent Firewall to Detect Novel Attacks - Pairing Detecting Malicious Patterns with Anomaly Detection. "**, Proc. of Conference on Information Security for South Africa (ISSA2002), Gauteng, South Africa, July 2002.

13. InSeon and Ulrich Ultes-Nitsche, **"An Integrated Network Security Approach : Pairing Detecting Malicious Patterns with Anomaly Detection."**, Proc. of Conference on Korean Science and Engineering Association in UK(KSEAUK2002). Surry University,Guildford, Surrey, UK. July, 2002.