# Modeling the Spread and Prevention of Malicious Mobile Code Via Simulation

CS-2004-15

by
Christopher Brian Shirey

Bachelor of Science
Computer Science
Florida Institute of Technology
2000

A thesis submitted to
Florida Institute of Technology
in partial fulfillment of the requirement for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
December, 2004

We the undersigned committee hereby recommended
that the attached document be accepted as fulfilling
in part the requirements for the degree of
Master of Science of Computer Science.

"Modeling the Spread and Prevention of Malicious Mobile Code Via Simulation"
a thesis written by Christopher Brian Shirey

Richard Ford, Ph.D.
Research Professor, Computer Science
Committee Chairperson

William D. Shoaff, Ph.D.
Associate Professor and Department Head, Computer Science

William Allen, Ph.D.
Assistant Professor, Computer Science

Muzaffar A. Shaikh, Ph.D.
Professor and Department Head, Engineering Systems

## Abstract

Title: Modeling the Spread and Prevention of Malicious Mobile Code
Via Simulation
Author: Shirey, Christopher Brian
Major Advisor: Richard Ford, Ph.D.

Malicious mobile code causes billions of dollars every year in damages, and that cost keeps increasing. Traditional signature-based anti-virus software is a reactive solution that can not detect fast spreading malicious code quickly enough to prevent widespread infection. If we hope to prevent widespread infection of future malicious mobile code, new prevention techniques must be developed that either stop a new infection completely, or at least limit the spread until signature-based anti-virus software can be updated.

Simulators exist that model the spread of malicious mobile code, but none currently exists that can efficiently model host-based and network-based spread prevention techniques and the effect that those techniques have on the spread of the infection. This thesis presents Hephaestus, which is a new simulator framework designed to meet these requirements and be flexible enough to meet future requirements. This thesis also presents the results of four experiments: one that models spread with no prevention techniques applied, one that models the effects of a monoculture on spread, one that models the effect of lossy detection, and one that shows the effects of tarpits.

# Contents

# List of Figures

# List of Tables

## Dedication

Waiting for 18 months which include pregnancy and a newborn for a project to be complete while our toddler is crying and needy can test the limits of one's patience. Throughout this time though, my wife's support for me and for this work has been unwavering, and so the following work is dedicated to my wife Kriste, without whose love, patience, and understanding this would not have been possible.

## Acknowledgments

Throughout the long and frustrating process of writing this thesis, many people have helped support and encourage me to continue, and I can never thank them enough. In particular, Dr. Richard Ford deserves special thanks for strongly encouraging me to push for excellence in this work rather than giving up prematurely when I got frustrated. This thesis would be much-lacking and poor without his motivation and support.

The work done on this thesis was funded by a grant from the Cisco Critical Infrastructure Assurance Group, and so I thank them sincerely. I have also received great support from Security Innovation. They have sponsored my graduate studies and have always been understanding and supportive. In particular Jason Taylor and Fred Orlando, my direct supervisors over the course of writing this thesis, have been nothing but encouraging. A special thanks also is extended to Helayne Ray, whose proofreading greatly improved the quality of this thesis.

# Chapter 1

# Introduction

This thesis explores the current state of malicious code modeling as it applies to prevention of widespread infection. We extend this prior research to provide a new framework for simulating the effects of various prevention techniques on the spread of malicious code. We call this new framework Hephaestus, named for the chief blacksmith of Greek gods who, among other things, created spontaneously operating automaton (anvils and bellows in his case) to help do his work for him[1].

## 1.1 Motivation

For approximately two decades malicious applications such as viruses have been damaging computers and data, and have been frustrating users of those computers. Last year, the total cost of viruses exceeded $50 billion[2]. Anti-virus software has become widely used, especially in large corporations, but the problem of stopping the malicious applications before they infect machines or damage valuable data continues to grow. Clearly, anti-virus software alone is not enough to solve the damaging effects of these applications.

New techniques can be developed, including proactive techniques, to help alleviate the problem, but there are issues that need to be addressed before those techniques can be installed and executed on the end-users' machines. The main issue is that a reliable method of testing the new technique needs to be developed. Solving this problem is not simple because the best way to test a new prevention technique is to apply it to real viruses on real networks. This is not a scenario most companies or users would appreciate, especially if the new technique does not work. The technique could be applied to a network disconnected from the Internet, but that network will never be as complex as the Internet, and the cost of cleaning and maintaining that network would be high. Even if the technique was tested in a network disconnected from the

Internet, the problem of verifying that the new technique significantly slows or stops the spread of a virus over a larger network or even the Internet would still remain.

By studying the spread of previously released malicious applications over real networks, researchers have found ways to predict the spread of similar future malicious applications. This can be accomplished using one of two different approaches - through using computer simulation or via analytical methods. To this point, none of these models have provided an efficient way to simulate multiple types of malicious applications over large networks while also simulating the effects of applied prevention techniques.

## 1.2 Problem Statement and Our Approach

The central problem that we address in this thesis is the lack of an efficient way to simulate multiple types of malicious applications over large networks that is also capable of simulating the effects of applied prevention techniques.

The approach we have taken to solve this problem is to create a discrete-time model, Monte Carlo, mixed abstraction level malicious application simulation framework. This framework is easy to use and yet powerful and flexible enough to test the efficiency of spread prevention techniques without use of an actual network, disconnected or otherwise.

## 1.3 Brief Summary of Results and Contributions

We have developed a framework capable of solving the aforementioned problem. We have managed to do so in a way that is both easy to use and yet powerful and flexible enough to simulate various prevention techniques.

We have executed various experiments to test the ability of the framework, and have gone to great lengths to describe how others could perform different experiments using the framework. As an additional contribution we have created a tool that can be used by educators to instruct students about how to think about the spread of malicious applications and ways to prevent that spread. By providing this, if we have enabled a new generation of students, future researchers, and future industry leaders to think practically about ways to help solve the problem of spreading of damaging malicious applications, then our contributions will have been greater than we could have hoped for.

## 1.4 Organization of the Thesis

In Chapter 2 we describe how previous researchers have thought of malicious code spread and how it can be modeled or simulated. We describe what the major research has revealed about the way malicious code spreads, and also describe some hurdles that have yet to be cleared.

In Chapter 3 we introduce and explain the Hephaestus simulation framework developed to allow researchers to be able to simulate the spread of malicious code as well as test potential spread prevention techniques. This framework is simple enough to be used by professors as a teaching tool, yet powerful and flexible enough to be used in industry to discover novel spread prevention techniques.

The Hephaestus simulation framework allows powerful experiments to be run easily. Chapter 4 shows how to run these experiments using Hephaestus and also shows how to examine and interpret the results of those experiments.

Despite the effort put forth in the development of Hephaestus, there is still much work that can be done to enhance its capabilities further. Chapter 5 outlines these future development tasks.

In Chapter 6 we present a summary of our work. We then conclude with two appendix chapters. In Appendix A, we provide proof that the random number generator the framework is using is in fact a very good random number generator. In Appendix B, we provide full source code to an actor library for readers to use as a reference when adding functionality to the Hephaestus framework.

# Chapter 2

# Previous Methods of Modeling the Spread of Malicious Code

In 1981, a virus known as Elk Cloner began spreading on Apple II floppy disks[3]. Early viruses such as Elk Cloner did not cause much financial damage to the computers and organizations that had become infected. Since that time millions of new computers became Internet-enabled and thousands of new viruses have been released. Many of these viruses have been designed to deliberately damage the machines they infect by deleting or corrupting files and data from the machines. These damages, intentional or otherwise, have become increasingly expensive over time. Trend Micro reported in early 2004 that the financial impact of all virus infections in 2001 totaled \$13 billion. In 2002 and 2003 those numbers grew to between \$20 and \$30 billion for 2002 and \$55 billion for 2003[2]. The Code Red virus alone had cost an estimated \$1.2 billion within the first 2 weeks of its release in July 2001 according to USA Today[4].

Due to the cost of these viruses the first line of defense against these malicious programs has become anti-virus software that will prevent infection by known viruses and clean old infections off the users' machines. However, despite the fact that most Internet users have signature-based anti-virus software installed on their computers, the cost of the viruses each year continues to climb. This continued increase is a strong indication that anti-virus products alone are not enough to protect Internet-enabled machines. The primary reason that anti-virus software is not sufficient to prevent infections is that it only detects existing viruses. It does this by matching unique signatures against files on the target machine. These signatures take some time to develop and distribute to the users' machines. As a result no newly created virus will be immediately detected and stopped by the software until the company that produces the anti-virus software can produce a signature for the virus and distribute it to the users' machines. Clearly, new methods of protecting users and machines against malicious code need to be created.

These new methods can be designed to protect a potentially infected machine by placing software on the machine directly, but other options exist as well. Protection methods that are distributed in nature could also be developed, protecting entire networks from attack. One problem with inventing new distributed prevention techniques is that it can be very difficult to test these new ideas without the cost of a creating and repeatedly cleaning a controlled environment in which to let viruses loose[5]. Prevention measures that are installed on the end-users' computers are much easier to test than distributed ones since there is only a single machine involved rather than a network of machines. If we want to be able to test new distributed spread prevention techniques without actually releasing viruses in a test network we need to be able to model and simulate the ways viruses spread in general. Once they are developed those models and simulations can be applied to our new prevention techniques. As an added benefit, creating the models and simulations allows researchers to predict global infrastructure failures when the Internet is exposed to fast moving viruses and worms[6]. Many attempts at modeling and simulating computer viruses have already been made by various researchers. This chapter details those attempts and analyzes the shortcomings that these models and simulations have if they are used to discover and test malicious code spread prevention techniques.

## 2.1 Analytical Models

Much of the research previously done on virus spread has primarily been done using analytical models and numerical computations for describing the rate which a virus will spread. These models are then implemented in computer simulations and compared against real-world virus spread data to determine a particular model's accuracy.

### 2.1.1 Early Work

Some of the earliest work in modeling the spread of viruses was done by Fred Cohen. He realized and demonstrated that infection can spread to the transitive closure of the information flow between systems. In other words if computer $A$ can infect computer $B$, and computer $B$ can infect computer $C$, then computer $A$ can infect computer $C$ indirectly[7][8]. He also stated that the only systems with potential for complete protection from a viral attack are machines with "limited transitivity and limited sharing, systems with no sharing, and systems without general interpretation of information (Turing capability)"[7]. Gleissner created a model of spread on a multi-user system and was able to verify that spread does in fact apply to the transitive closure of the information flow, and also showed that this spread could happen at an exponential growth rate[8][9]. Peter Tippett suggested later that computer viruses could spread at an exponential rate since many other population models exhibit similar growth patterns, but he never presented a model to back up his claims, and the fact that real viruses do not conform to an exponential growth rate has caused many to doubt his claims[8][10].

In 1990, Solomon created a model of virus spread that contradicted the idea that viruses can spread exponentially between systems. He explained that there are three factors that can affect the probability that a given virus has infected a given machine, and those factors are listed below[11]:

1. The percentage of currently infected individuals
2. The readiness with which the virus under consideration can replicate (called infectivity)
3. The degree to which the machine in question has contact with the population of computers

Solomon also listed 2 factors that affect the percentage of currently infected individuals, listed below:

1. The rate at which computers are becoming infected
2. The length of time that they stay infected

Based on these factors Solomon developed an equation that showed that the rate of infection in previously uninfected machines is proportional to the number of currently infected machines, the number of uninfected machines, and to the probability of infection. The equation also shows that the rate of infections being cleaned is proportional to the number of infected machines and to the probability of detection. The equation follows:

$$\frac{dp}{dt} = P(1 - P)I - PD \tag{2.1}$$

In equation 2.1 $P$ is the percentage of machines that are infected with the virus, $I$ is the probability that a machine will become infected by contact with another infected machine, and $D$ is the probability that the virus is detected[11].

This model produces some interesting results. If we set $\frac{dp}{dt}$ to 0 we find that either there is no infection and no spread of infection of the virus or that $P$ = 1 - $D/I$. This means that if $D$ is less than or equal to $I$, the spread of the infection will approach 0 and die out, but if $I$ is greater than $D$ the virus will continue to spread until eventually all the potentially infected machines are infected[11].

### 2.1.2 The SIR and SIS Models

Epidemiological models have often been used to attempt to predict the spread of malicious programs in computer systems. Two of the most commonly used epidemiological models are the Susceptible-Infected-Removed (SIR) and the Susceptible-Infected-Susceptible (SIS) models. In the SIR model a system can become infected once, and once the infection is removed it can never again become infected. In the SIS model a system can repeatedly become infected and have the infection removed[12].

Both the SIR and the SIS models are homogeneous, meaning that an infected individual machine is equally likely to infect any of the susceptible machines. This model works fine for many biological viruses like influenza in which the disease is transmitted via casual contact but the analogy between biological systems and computer systems does not hold as well for computer viruses since program sharing is not homogeneous. Computer users generally only share programs and files with a limited number of people, generally people they know. The users never have contact with the vast majority of the world's computer users and therefore accurate models of computer virus spread need to have the ability to model inhomogeneous spread[8].

#### The KW Directed Graph SIS Model

Kephart and White saw the discrepancy between homogeneous biological systems and inhomogeneous computer networks and developed a model based on the SIS epidemiological model applied to directed graphs in 1994[8]. In this model, called the KW model for the rest of this paper, each computer system is represented as a node on a graph. Directed edges from a node $n$ to other nodes represent the computers that can be infected by $n$. Each node has a property which specifies the rate that an infection can be cured and removed. Similarly each edge has a property that specifies the infection rate from one node to the node on the other end of the edge[8]. After developing the model and then removing the details of the edges, their results were in agreement with the classical SIS model, which is to be expected but was necessary to verify in order to ensure their model was correct.

The development and testing of the KW model provided us with some fundamental and important results. Although Cohen proved that a perfect defense against computer viruses is impossible, the KW model shows that perfection is also not necessary in order to prevent the widespread infection of a computer virus if the rate of detection and removal of infection is high relative to the rate at which the virus propagates from one computer to another. This is shown by proving that the known fact that a biological epidemic can only occur if the infection rate exceeds a finite critical threshold also applies to computer viruses as well. Kephart and White also discovered that the topology of the graph has a profound effect on the ability of a virus to spread, and this could not have been shown through modeling only homogeneous systems[8].

#### Wang and Wang's Application of Timing Parameters to the SIR and SIS Models

The models presented to this point have abstracted away two potentially important timing variables - the *infection delay* and *user vigilance*. Infection delay is simply defined as the amount of time between the instant the machine is infected until it begins attempting to infect other machines. User vigilance is defined as the amount of time users will be more cautious of potential viruses once they have been infected. Introducing these variables simulates real-world scenarios better than models that ignore this variable because after a user's machine has become infected, they are much more wary of taking actions similar to those that allowed their computer to become infected previously. This is especially true of viruses like email viruses which require some user action before the infection will occur. In the classic SIR and SIS models as well as in the KW model, these delays are both assumed to be zero. In other words, as soon as a computer becomes infected in these models, that computer *immediately* begins infecting other computers. This is not the case in real-world computer systems, so to better model reality Wang and Wang enhanced the KW model taking into account these parameters to determine what effect they would have on the rate of spread of a virus[12].

By adding these two parameters to the models, two important results that validate Wang and Wang's desire to include the timing parameters in the models are revealed. First, adding the infection delay increases the epidemic threshold of the virus, meaning that modeling the infection delay makes an infection die out before reaching epidemic levels more often than without modeling it. Secondly, adding the user vigilance measure to the model does not change the epidemic threshold in either direction. The spread of infection is slowed by including either of these parameters in the model, but only the infection delay has any impact on the epidemic threshold[12].

### 2.1.3 The Kill Signal Model

Although Kephart and White's previous work ([8]) had revealed some important discoveries, the KW model did not always accurately predict or simulate the actual spread of a virus in the real world. There was such a lack of information about the prevalence of real world viruses that highly respected experts incorrectly estimated the spread of the Michelangelo virus in 1992 by more than three orders of magnitude[13]. Kephart and White then created another spread model to help more accurately estimate the prevalence of viruses and give anti-virus software vendors new approaches to slowing the spread. The new model is known as the Kill Signal Model.

In the previous epidemiological models, a virus is cured independently of the other infected systems. But what would happen if a user noticed that the machine they were using was infected and told several friends or coworkers about the need for them to update and run their anti-virus software to scan for that virus? If they followed the first user's advice and the virus got detected and prevented before their machines were infected, the virus would have a much harder time spreading to epidemic levels. That scenario is the reasoning behind the Kill Signal Model, and Kephart and White refer the first user's warning to the others about the virus as the "kill signal"[13]. Kephart and White tested this model two different ways - one where the kill signal is delivered and acted upon more quickly than the virus itself spreads and one where the kill signal is not delivered and acted upon instantaneously.

When transmitted instantaneously, Kephart and White discovered that a relatively low ratio of computers need to receive and act upon the kill signal in order to prevent epidemic spread. In this scenario, as soon as the first machine became cured of the infection, it immediately sends the kill signals to its immediate neighbors on the directed graphs of the KW model. Upon receipt of the kill signal, the neighbors would become cured and send the kill signal to all of their immediate neighbors and so on. In their experiments Kephart and White found that with a received kill signal probability of less than 0.25, the epidemic probability did not change significantly, but once it passed the 0.25 threshold, the epidemic probability abruptly drops off to zero. This means that if 3 out of a node's 10 neighbors receive and act upon the kill signal, extinction of the virus in inevitable[13].

If the kill signal is not transmitted instantaneously, the kill signal can be thought of as an anti-virus epidemic and the kill signal can then be modeled and treated just like a regular virus in terms of the rate of spread except that it has no intrinsic epidemic threshold and as such can not die out until desired to do so. Once the kill signal is received on a machine, the machine will never become infected again (as it already knows about the virus) and so the model ultimately reduces to the SIR model[13].

### 2.1.4 The PSIDR Model

One thing missing from the other epidemiological models is that none of them take into account the effect that installed, properly working anti-virus software has on the spread of a virus. Williamson and Léveillé developed the Progressive Susceptible Infectious Detected Removed (PSIDR) model to address this[14].

In the PSIDR model machines are assumed to be in one of four states - Susceptible, Infectious, Detected (a state where the virus has been detected and is no longer able to spread further), and Removed. Once the virus signature is distributed to a given machine, the machine becomes Removed if its state prior to receipt of the signature was Susceptible. If the prior state was Infected, the machine then moves to the Detected state. The rate of virus spread in this model is in essence a race between the virus spread and the anti-virus signature spread. The signature spread has an initial disadvantage, however, because of the lag time from virus outbreak until the virus signature is produced and started being distributed.

The result of running experiments using this model is that distributing signatures more quickly will lead to a smaller outbreak of the virus. The authors conclude that clients should poll the signature servers more often than they currently do, even to the point of suggesting that the servers should try to push the signatures to its clients to further enhance the speed of the process of signature distribution[14].

### 2.1.5 The Random Constant Spread Model

After Code Red II was released in 2001, Staniford, Paxson, and Weaver developed a model to describe and predict the rate which worms like Code Red would spread. These are worms that repeatedly attempt to infect machines at random Internet addresses. They found the following equation to fit the observed spread well

$$a = \frac{e^{K(t-T)}}{1 + e^{K(t-T)}} \tag{2.2}$$

where $a$ is the proportion of machines already infected, $K$ is the initial infection rate, $T$ is the time that the infection begins to spread, and $t$ is the current time[15]. The graph of actual scans versus the number predicted by this model is shown in figure 2.1.

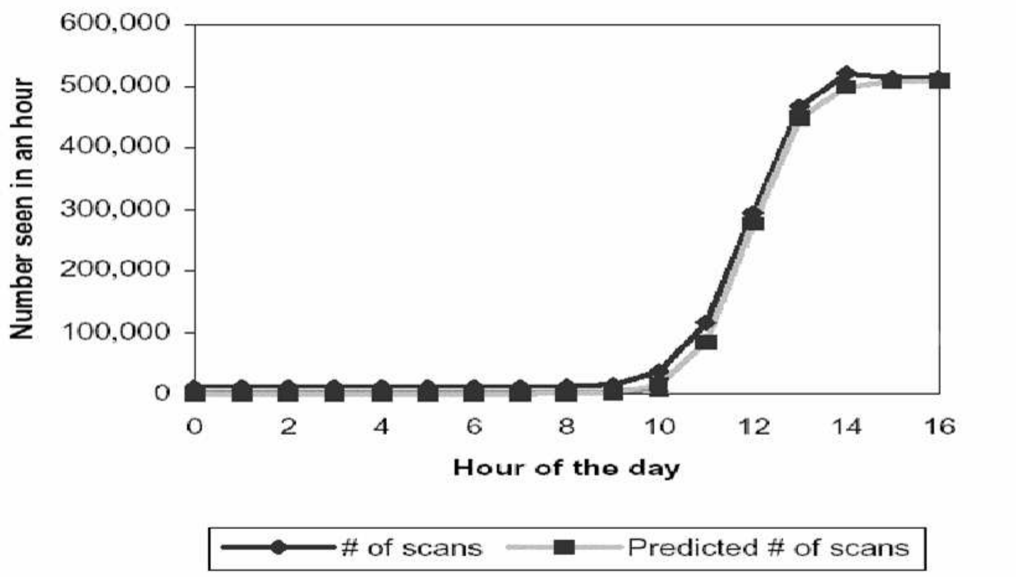


Figure 2.1: The Random Constant Spread Model Accuracy for Code Red

Serazzi and Zanero had studied the Random Constant Spread (RCS) Model, but when the SQL Slammer worm appeared they noticed a problem. They noticed that the RCS model fit the actual data well towards the beginning of the outbreak, but as time went on the RCS model did not predict the spread of SQL Slammer as well as they had expected (Figure 2.2)[6].

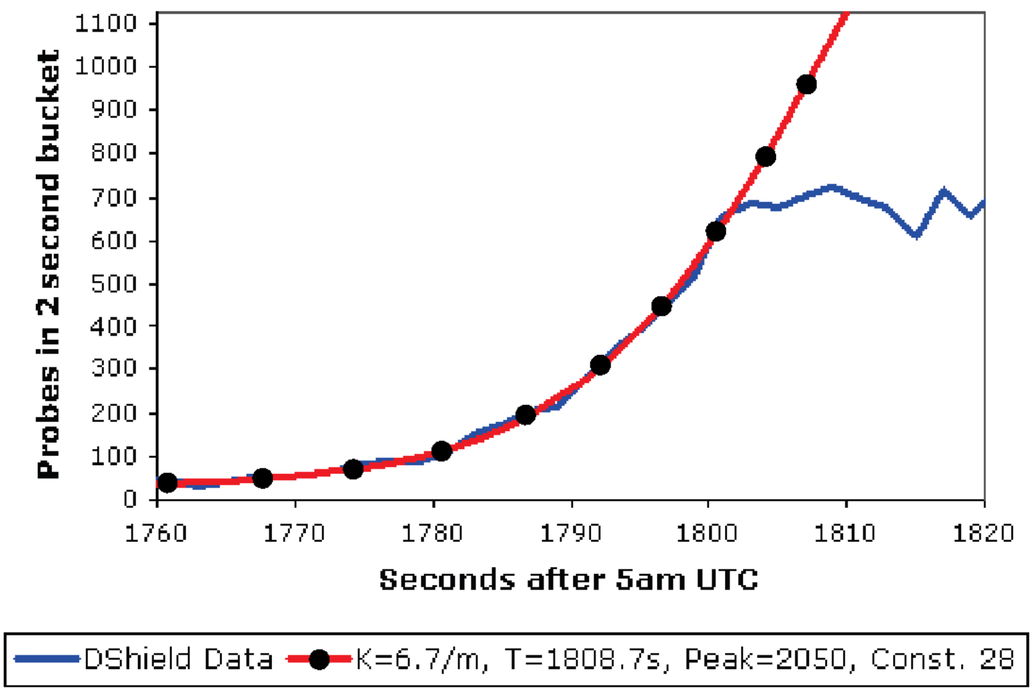


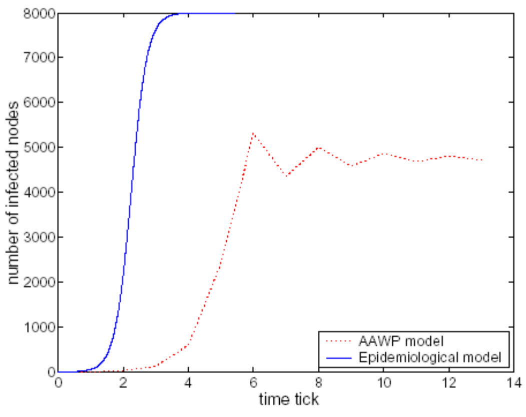
Figure 2.2: The Random Constant Spread Model Accuracy for SQL Slammer

They hypothesized that this discrepancy was due to the network becoming overloaded with traffic and so the network had a bottleneck that prevented the spread from following the RCS curve. Serazzi and Zanero then updated the RCS model to take network bandwidth into account and verified that their suspicions were correct[6].

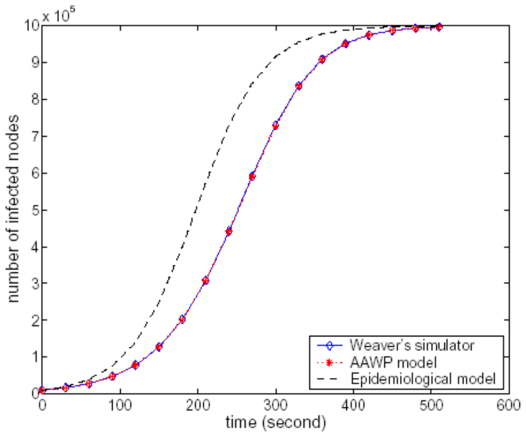
### 2.1.6 The AAWP Model

Chen, Gao, and Kwait developed the Analytical Active Worm Propagation (AAWP) model after noticing that the epidemiological models still fell short in a few areas. First the epidemiological models use a continuous time differential equation, meaning a newly-infected computer starts infecting other machines before the computer itself is fully infected. This can produce inaccurate results because it will show the virus spreading more quickly than in a real-world scenario. Secondly, the epidemiological models do not take into account either a patching rate or the time it takes to infect a machine. Note that the patching rate is not necessarily the same as the death rate of a virus. The patching rate is the rate at which machines become permanently cured of possible infection from a particular virus, whereas the death rate is the rate at which a system has the virus removed, and this removal may not be permanent. Lastly, the epidemiological models do not allow for the case where a single machine is being infected by two or more infected machines at the same time[16]. The AAWP model was developed to help alleviate these seen deficiencies and be a model used to predict the spread of random scanning viruses. Although the AAWP model is an analytical model, these realizations argue for a discrete time Monte Carlo simulation approach to spread modeling as simulating the spread in this way easily can alleviate their concerns.

In addition to adding the patching rate, the time it takes for a machine to become infected, and the ability to have multiple machines infecting the same machine simultaneously, the AAWP introduces a discrete time system in which all events occur in one or more discrete time blocks rather than in fractions of seconds. By making these changes, the results differed substantially compared to the epidemiological models when attempting to model Code Red II. The results are shown in Figure 2.3.



(a) All cases are for 10,000 vulnerable machines, a hitlist with 1 entry, a scanning rate of 2147500 scans/time tick or a birth rate of 5 /time tick and a death rate of 1 /time tick. No patching and a time period of 1 time tick to complete infection for the AAWP model.



(b) All cases are for 1,000,000 vulnerable machines, a hitlist with 10,000 entries and a scanning rate of 100 scans/second. A time period of 30 seconds to complete infection for Weaver's simulator and the AAWP model. A death rate of zero for both the AAWP model and the Epidemiological model. No patching for the AAWP model.

Figure 2.3: The AAWP Compared to the Epidemiological Model

The results of the experiments run using the AAWP model match very closely to the actual reported spread of Code Red II. It also helps to solve the "mystery" of virus research presented by White that the epidemiological models predict far more widespread infection of many viruses than have actually been seen in real world scenarios[17].

## 2.2 Monte Carlo Simulations

A non-analytical approach to modeling and predicting virus spread is by simulation. This involves developing a computer program that tries to closely imitate real-world scenarios. Monte Carlo simulations are simulations where values for unpredictable variables are chosen by generating pseudorandom numbers when that value is needed. The term Monte Carlo simulations comes from Monte Carlo, Monaco, where casinos with games of chance are a main attraction[18]. This section describes several previously-developed Monte Carlo simulators.

### 2.2.1 The Weaver Simulator

The Weaver Simulator is a rather simple simulator written by Nicolas Weaver to demonstrate the results of his research into the theoretical Warhol worm. Warhol worms are defined by Weaver as "hyper-virulent active worms, capable of infecting all vulnerable hosts in approximately 15 minutes to an hour"[19]. It employs a fully connected network where there is no network hardware to consider and each connection has unlimited bandwidth. It allows infection attempts by three methods - random scanning, permutation scanning, and partitioned permutation scanning. This allows for the simulation of worms that start by infecting a predefined "hit list" of machines before randomly attempting to infect the rest of the network. Several options are configurable by the user such as the number of vulnerable machines, the size of the initial population, the number of scans per second, and the time it takes to infect a machine[19].

### 2.2.2 NWS

The Network Worm Simulator by Bruce Edigar is a framework for simulating network worms, their spread, and their effect on the infected machines. The framework is written in Perl and allows each worm to be written in Perl. It is different from other simulators in the respect that user-specified Perl code can be set to be executed on a machine when that machine becomes infected to get a more realistic picture of the damage a simulated worm can induce. Each node is directly connected to every other node on the network. There is no network hardware or routes that the virus must pass through in order to get from one node to another in the simulator. It uses either the SIS or SIR model epidemiological model to switch between infection states depending on which the user of the simulator chooses to use[20].

### 2.2.3 DDoSVax

DDoSVax is an ongoing project by the Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich. The purpose of the project is to detect and prevent both the infection phase and the denial of service phase of Distributed Denial Of Service (DDoS) attacks in real-time[21].

They have written a simulator to simulate DDoS attacks and their prevention measures. Their simulator plots the number of infected machines vs time as well as the infection rate over time by default. It can model random scanning, hit lists, and local forced scanning (scanning local nodes more often than external nodes). It is written in Perl and does its plotting using GnuPlot. The simulator has a large number of configurable parameters to change the setup being simulated. Some of these parameters are total number of machines, number of initially infected machines, a hit list, and TCP timeout.

The primary weakness of DDoSVax for our purposes is that the project lacks the capability to simulate prevention measures on the target machines. The project's prevention measures are targeted towards network hardware rather than attacked machines. Also, much of the simulator's resources must be devoted to generating and tracking the network traffic rather than the effects of the malicious application, as the simulator is designed to analyze network traffic at the packet level.

### 2.2.4 SSF.App.Worm

SSF.App.Worm is an interface that works on top of the Scalable Simulation Framework Network Models (SSFNet)[22]. It models the spread of specific worms such as CodeRed or SQLSlammer over the Internet. It uses 2 abstraction levels to simulate the spread - a macroscopic level using various epidemiologic models from biology (the SIR model in particular) and a microscopic level to simulate a worm's interaction with network infrastructure. SSF.App.Worm assumes that the worm spreads by randomly targeting other machines on the network to infect. There are also many configurable parameters that can be specified in the simulations such as the total number of susceptible machines, the number of initially infected machines, and a removal function to specify when infected machines can be recovered from infection[23].

This is a substantial improvement over the previous simulators in that it models the spread of worms over the Internet. This gives good results when simulating the speed a given worm spread. Since it simulates traffic over network hardware such as routers and bridges, SSF.App.Worm could be extended to simulate prevention measures installed on that hardware. The user of the simulation could then see what effect those prevention measures would have on real-world viruses like SQLSlammer and CodeRed.

As with DDoSVax, all prevention measures in SSF.App.Worm must be implemented in the network hardware, making it impossible to simulate host-based prevention measures. Also, due to the fact that they simulate the traffic between each piece of network hardware as well as to the end user's machines, much of the simulator's resources must be devoted to generating and tracking the network traffic rather than the effects of the malicious application itself.

### 2.2.5 Shortcomings of the Previous Simulators

Although each of the simulators are important and useful, none of them are perfect. Some of these shortcomings are addressed in this thesis, while other shortcomings are left to be addressed in future work.

The Weaver Simulator has no ability to simulate prevention measures, either network or host based. It also has no ability to define nodes other than susceptible, infected, or not susceptible such as sensor nodes, tar pits, or even unreachable machines[19].

For NWS, the epidemiological models used do not hold up when applied to computer systems, as has been shown. There is also no way to simulate prevention measures on each host. Since the framework and simulated worms are all written in the scripting language Perl, Edigar points out that the performance of the simulator will never be enough to simulate the entire Internet[20].

For DDoSVax, the project's goal includes a statement that they will research possible semi-automatic countermeasures to DDoS attacks[21], but all countermeasures are implemented in the network hardware making it impossible to simulate host-based prevention techniques. Also, since they are targeting DDoS attacks, they focus much of their resources into analyzing the traffic generated by the DDoS worms. This approach limits the use of this simulator for more general purpose worms since the network traffic is not necessary to simulate the spread of malicious applications over a network.

SSF.App.Worm also has some shortcomings in the way prevention measures can be simulated as well as modeling worm spread in general. First, as has been noted the SIR epidemiological model that the simulator uses for high level infection simulation does not hold up well for non-biological systems such as computer networks. Secondly, since SSFNet simulates the actual traffic generated by the worm, fast spreading worms such as SQLSlammer and CodeRed generate enough traffic to make the resources needed for the simulation more than a modest machine can handle[23][24]. Modeling the network routes that a worm takes to infect another machine is important in many cases, but generating the traffic between the network hardware is often unnecessary in simulation. Finally, any prevention measures simulated would need to be simulated on network hardware rather than on the host machines that are being infected. Having the ability to model prevention measures on the host opens up a whole new set of experiments such as lossy detection (see section 4.3) that would be difficult or impractical to apply to network hardware.

# Chapter 3

# Hephaestus

While studying the previous simulators, especially due to the shortcomings listed in section 2.2.5, it became clear that a new simulation framework was needed to efficiently fulfill each of the following three requirements:

1. Needs to be able to model malicious code spread over a network
2. Needs to be able to model custom host based prevention techniques
3. Needs to be able to model custom network based prevention techniques

Several of the previous simulators had the ability to fulfill two of these requirements, but none could fulfill all of them. Hephaestus is a new simulation framework designed to fulfill all these requirements and be extensible to future requirements.

## 3.1 Overview of Hephaestus

Hephaestus is a discrete time model, mixed-level abstraction, Monte Carlo simulation framework designed to simulate how mobile code can spread across a large network. In theory, the network can be consisted of up to $2^{32}$ machines. In practice, however, the resources needed to model a network of this size are more than a modern workstation can hope to provide. It models nodes in a network and allows configuration of each node. Using this method, we can learn more about how mobile code currently spreads, determine the limits of how quickly a given infection can propogate through a network, how widespread the infection can become, and most importantly what measures can be taken to prevent epidemic or pandemic outbreaks until reactive solutions can be put in place such as anti-virus signature updates.

The Hephaestus framework is composed of several different components, including the simulation engine component HephaestusEngine. The engine cycles through the set of infected nodes and for each of the infected nodes the engine randomly selects another node to try to infect. This node could be infected already, it could be uninfectable, or it could be configured in one of many other ways. After all the infected nodes have had a chance to infect another node, the entire cycle begins again, potentially with newly infected nodes. Each of these cycles is called a “time step” and the number of time steps that the simulation goes through is configured in the simulator. After all the time steps have been executed the simulation ends and the results are displayed or logged to a file.

By studying the results of experiments run using Hephaestus we can learn about, model, and apply measures designed to slow or stop the spread of a given category of malicious code. Hephaestus also allows these countermeasures to be easily modeled and have their effects studied. To allow this, the nodes involved in the simulator are highly configurable. As part of this configuration the user can specify whether or not a node can ever be infected, whether it can be connected to, and many other options. By changing these parameters we can simulate how a machine with prevention measures applied can help slow or stop a network infection completely.

## 3.2 Components Making Up the Hephaestus Framework

Hephaestus consists of several components that work together to make the simulator framework. Below is a brief description of each of them.

- HephaestusEngine - This is the driving force behind the entire simulator and is the most important component. All other components connect and interact with the engine to set up and get the results of the experiment to report.

- Views - This is an API used as a window into the actions taken by the engine. All events that happen in the engine get reported to registered views for display to the user.

- Command Line Interface - A console-based front-end to the simulator. Contains several logging options and is fully configurable.

- Win32 User Interface - A Windows front-end to the simulator. Contains all the options that the Command Line Interface supports, and adds graphs to help the user visualize the results of the experiment.

- ScPl - The plotting library used in the Win32 User Interface to display the visual representations of the simulation results.
- Actor Libraries - Each of these Windows Dlls represents a different type of actor node in the simulation. These are configurable, and can be plugged into the simulator by specifying them in the configuration files passed to the simulator.

A diagram showing where each of these pieces fit into the framework is shown in figure 3.1.

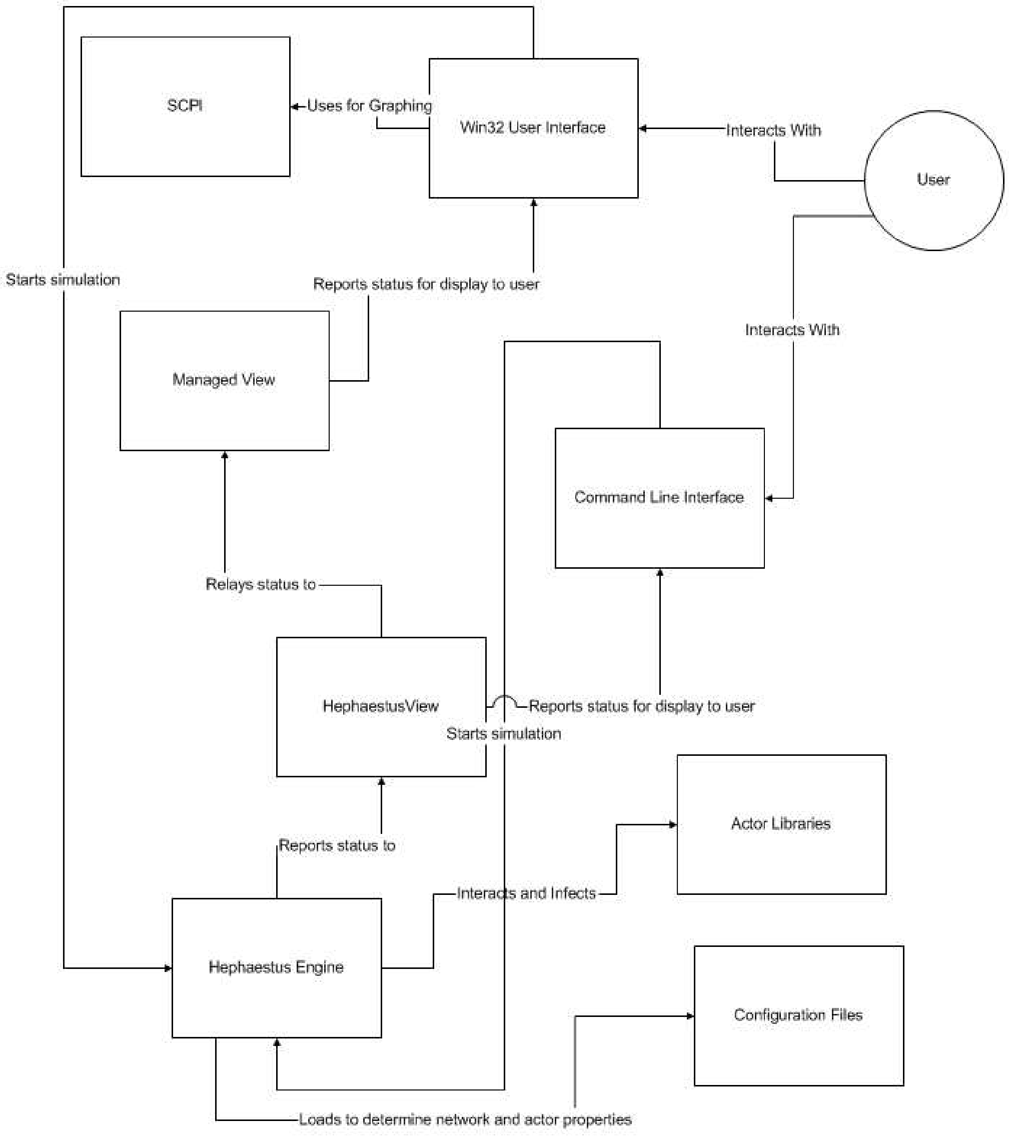


Figure 3.1: Components of the Hephaestus Simulation Framework

### 3.2.1 HephaestusEngine

The HephaestusEngine component is the heart of the Hephaestus simulator framework. It controls the nodes' interactions during each timestep of the simulation and reports results to components that are interested. One detail to keep in mind is that at each timestep, each infected node gets to try to connect to another node. Which node gets targeted is determined by a pseudorandom number generator (a Mersenne Twister implementation, see Appendix A), so currently all infection is done at random. There is no way to simulate an "intelligent" node that targets nodes to infect in any way other than completely at random in the current version of Hephaestus, but different targeting algorithms can easily be added in future versions.

Below is sample C++ code to instantiate an instance of the engine. The comments above each line will describe how to use the methods in that line.

```
// HephaestusEngine is the class to instantiate. The parameter to
// the constructor is the configuration file to use for the simulation
HephaestusEngine *engine = new HephaestusEngine("c:\\actorlist.txt");
if (!engine)
  return -1;

HephaestusCLISimpleView *view = new HephaestusCLISimpleView();
if (!view)
{
  delete engine;
  return -1;
}
// registers a view for results notification. More on this later in the page.
engine->RegisterView(view);

// sets the number of nodes infected at the start of the simulation
engine->SetNumInfected(1);

// sets the maximum number of time steps the simulation will execute for
engine->SetTimeSteps(100);

// sets the seed for the Mersenne Twister random number generator
engine->SetRandSeed((unsigned long)time(NULL));

// sets the number of infected nodes that when reached will cause the
// simulation to stop executing. Set to 0 for no threshold.
engine->SetThreshold(10000);

// sets the number of time steps that if no new nodes are infected will cause
// the simulation to stop executing.  Set to 0 for no stability threshold
engine->SetStabilityThreshold(10);

// starts the simulation. all results are sent to any registered views. More
// on this later in the page.
engine->RunSimulation();

delete view;
delete engine;
```

### 3.2.2 Views

In the source code listed in section 3.2.1, there was mention of a "view" used to get the results of the engines actions. A view is an instance or subclass of 1 of 2 classes - one for pure Win32 code and one for managed (Microsoft .NET Common Runtime Library compatible) code. Users can register more than 1 view for each HephaestusEngine instance and each will get notified of all results. These two classes are "HephaestusView" and "ManagedView".

### HephaestusView

The HephaestusView class is part of the HephaestusEngine component. Subclassing this class and registering it with the engine will allow the user to get notified of results of the simulator as they happen. This class is for pure Win32 code. For managed code users should use the ManagedView class. Note that nearly every method in the HephaestusView class is pure virtual. This means that when a user subclasses this class, they have to provide an implementation method for each method in this class that is pure virtual, even if the implementation method does not do anything. The definition for this class is listed below.

```
class SimulatorEngine::HephaestusView
{
private:
public:
  HephaestusView();
  virtual ~HephaestusView();

  // called when the initial number of infected nodes are set. value is the number infected
  virtual void NumInfectedSet (unsigned long value) = 0;
  // called when the number of time steps is set. value is the number of time steps set
  virtual void TimeStepsSet (unsigned long value) = 0;
  // called when the random number seed is set. value is the seed value set
  virtual void RandSeedSet (unsigned long value) = 0;
  // called when the threshold value is set. value is the threshold value set
  virtual void ThresholdSet (unsigned long value) = 0;
  // called when the stability threshold value is set. value is the stability threshold
  // value set
  virtual void StabilityThresholdSet (unsigned long value) = 0;

  // called when the simulation has started
  virtual void SimulationStarted () = 0;
  // called when the simulation is over
  virtual void SimulationEnded () = 0;

  // called at the beginning of each timestep during the simulation. timestep is the
  // timestep value that was just started
  virtual void TimestepStarted (unsigned long timestep) = 0;
  // called at the end of each timestep during the simulation. timestep is the timestep
  // value that has just ended. totalInfected is the total number of infected nodes so far.
  // timestepInfected is the number of nodes that were infected in this timestep
  virtual void TimestepEnded (unsigned long timestep, unsigned long totalInfected,
      unsigned long timestepInfected) = 0;

  // called when it is a particular node's turn to connect/infect another node. node is the
  // node that has this focus
  virtual void NodeHasFocusPhaseStarted (NetworkNode *node) = 0;
  // called when a particular node's turn to connect/infect another node has ended. node is
  // the node that has lost this focus.
  virtual void NodeHasFocusPhaseEnded (NetworkNode *node) = 0;

  // called when the phase of each timestep has started where already opened sockets attempt
  // to connect to the nodes on the other end of the socket
  virtual void ReconnectOpenSocketsPhaseStarted () = 0;
  // called when the phase of each timestep has ended where already opened sockets attempt
  // to connect to the nodes on the other end of the socket
  virtual void ReconnectOpenSocketsPhaseEnded () = 0;
```

```
// called before the simulation starts when the engine is trying to create all the nodes
// for the simulation
virtual void NodeCreationPhaseStarted () = 0;
// called before the simulation starts a new node is created.  node is the node created
virtual void NodeCreated (NetworkNode *node) = 0;
// called before the simulation starts when the engine is done trying to create all the
// nodes for the simulation
virtual void NodeCreationPhaseEnded () = 0;

// called before the simulation starts when the engine has started trying to infect the
// initially infected nodes
virtual void InitialInfectionPhaseStarted () = 0;
// called before the simulation starts when the engine has infected a node. node is the
// node that was infected
virtual void InitialInfection (NetworkNode *node) = 0;
// called before the simulation starts when the engine has finished trying to infect the
// initially infected nodes
virtual void InitialInfectionPhaseEnded () = 0;

// called during the simulation in each timestep when the phase to infect nodes has
// started
virtual void InfectionPhaseStarted () = 0;
// called during the simulation in each timestep when an infection has been attempted.
// sendingNode is the node doing the infection attempt. receivingNode is the node to be
// infected. socket is the open socket between the 2 nodes. result is the result of the
// infection attempt.
virtual void InfectionAttempt (NetworkNode *sendingNode, NetworkNode *receivingNode,
    NetworkSocket *socket, InfectionResult result) = 0;
// called during the simulation in each timestep when the phase to infect nodes has ended
virtual void InfectionPhaseEnded () = 0;

// called during the simulation in each timestep when the phase to connect to a node has
// started. sendingNode is the node doing the connecting. receivingNode is the node being
// connected to.
virtual void NewConnectionPhaseStarted (NetworkNode *sendingNode,
    NetworkNode *receivingNode) = 0;
// called during the simulation in each timestep when the phase to connect to a node has
// ended. sendingNode is the node doing the connecting. receivingNode is the node being
// connected to.
virtual void NewConnectionPhaseEnded (NetworkNode *sendingNode,
    NetworkNode *receivingNode) = 0;

// called when a node has been successfully infected. node is the newly infected node
virtual void NodeWasInfected (NetworkNode *node) = 0;
// called when the infected threshold has been exceeded
virtual void ThresholdExceeded () = 0;
// called when the stability threshold has been exceeded
virtual void StabilityThresholdExceeded () = 0;
// called during the simulation in each timestep when a connection has been attempted.
// node1 is the node attempting the connection. node2 is the node being connected to.
// socket is the socket between the 2 nodes. socketTimeOpen is the length of time (in
// timesteps) that the socket has been open. connectionResult is the result of the
// attempt.
virtual void ConnectionAttempt (NetworkNode *node1, NetworkNode *node2,
    NetworkSocket *socket, unsigned short socketTimeOpen,
    ConnectionResult connectionResult) = 0;
// called when a new socket has been created between 2 nodes. socket is the socket
// created
virtual void SocketAdded (NetworkSocket *socket) = 0;
// called when a socket has been deleted between 2 nodes. socket is the socket deleted.
virtual void SocketDeleted (NetworkSocket *socket) = 0;
```

```
};
```

### ManagedView

This class is part of the Win32 User Interface and is HephaestusView's counterpart for managed code. Unlike HephaestusView users do not need to subclass this class, they can simply use it. Users also do not need to provide an implementation method for each of these delegates, only the ones they want to register for. To see what each of these delegates does, see the documentation for the corresponding HephaestusView method listed in section 3.2.2. Also notice that the ManagedView class is a singleton and has a private constructor. This means it can not be instantiated directly. To get a pointer to the instance of the class, use code like the following:

```
ManagedView *view = ManagedView::GetInstance();
```

The definition for the ManagedView class follows.

```
public __delegate void NumInfectedSetDelegate (unsigned long);
public __delegate void TimeStepsSetDelegate (unsigned long);
public __delegate void RandSeedSetDelegate (unsigned long);
public __delegate void ThresholdSetDelegate (unsigned long);
public __delegate void StabilityThresholdSetDelegate (unsigned long);
public __delegate void SimulationStartedDelegate ();
public __delegate void SimulationEndedDelegate ();
public __delegate void TimestepStartedDelegate (unsigned long);
public __delegate void TimestepEndedDelegate (unsigned long, unsigned long, unsigned long);
public __delegate void NodeHasFocusPhaseStartedDelegate (NetworkNode *);
public __delegate void NodeHasFocusPhaseEndedDelegate (NetworkNode *);
public __delegate void ReconnectOpenSocketsPhaseStartedDelegate ();
public __delegate void ReconnectOpenSocketsPhaseEndedDelegate ();
public __delegate void NodeCreationPhaseStartedDelegate ();
public __delegate void NodeCreatedDelegate (NetworkNode *);
public __delegate void NodeCreationPhaseEndedDelegate ();
public __delegate void InitialInfectionPhaseStartedDelegate ();
public __delegate void InitialInfectionDelegate (NetworkNode *);
public __delegate void InitialInfectionPhaseEndedDelegate ();
public __delegate void InfectionPhaseStartedDelegate ();
public __delegate void InfectionAttemptDelegate (NetworkNode *, NetworkNode *,
    NetworkSocket *, InfectionResult);
public __delegate void InfectionPhaseEndedDelegate ();
public __delegate void NewConnectionPhaseStartedDelegate (NetworkNode *, NetworkNode *);
public __delegate void NewConnectionPhaseEndedDelegate (NetworkNode *, NetworkNode *);
public __delegate void NodeWasInfectedDelegate (NetworkNode *);
public __delegate void ThresholdExceededDelegate ();
public __delegate void StabilityThresholdExceededDelegate ();
public __delegate void ConnectionAttemptDelegate (NetworkNode *, NetworkNode *,
    NetworkSocket *, unsigned short, ConnectionResult);
public __delegate void SocketAddedDelegate (NetworkSocket *);
public __delegate void SocketDeletedDelegate (NetworkSocket *);

public __gc class HephaestusGui::ManagedView
{
private:
  ManagedView ();
  ~ManagedView ();
  static ManagedView *instance = NULL;
  ManagedViewBase *base;
public:

  static ManagedView *GetInstance ();

  // gets the HephaestusView object that this class is based on.
  __property HephaestusView *get_UnmanagedView () { return base; }

  NumInfectedSetDelegate *onNumInfectedSet ;
  TimeStepsSetDelegate *onTimeStepsSet ;
  RandSeedSetDelegate *onRandSeedSet ;
  ThresholdSetDelegate *onThresholdSet ;
  StabilityThresholdSetDelegate *onStabilityThresholdSet ;
  SimulationStartedDelegate *onSimulationStarted ;
  SimulationEndedDelegate *onSimulationEnded ;
  TimestepStartedDelegate *onTimestepStarted ;
  TimestepEndedDelegate *onTimestepEnded ;
  NodeHasFocusPhaseStartedDelegate *onNodeHasFocusPhaseStarted ;
  NodeHasFocusPhaseEndedDelegate *onNodeHasFocusPhaseEnded ;
  ReconnectOpenSocketsPhaseStartedDelegate *onReconnectOpenSocketsPhaseStarted ;
  ReconnectOpenSocketsPhaseEndedDelegate *onReconnectOpenSocketsPhaseEnded ;
  NodeCreationPhaseStartedDelegate *onNodeCreationPhaseStarted ;
  NodeCreatedDelegate *onNodeCreated ;
  NodeCreationPhaseEndedDelegate *onNodeCreationPhaseEnded ;
  InitialInfectionPhaseStartedDelegate *onInitialInfectionPhaseStarted ;
  InitialInfectionDelegate *onInitialInfection ;
  InitialInfectionPhaseEndedDelegate *onInitialInfectionPhaseEnded ;
  InfectionPhaseStartedDelegate *onInfectionPhaseStarted ;
  InfectionAttemptDelegate *onInfectionAttempt ;
  InfectionPhaseEndedDelegate *onInfectionPhaseEnded ;
  NewConnectionPhaseStartedDelegate *onNewConnectionPhaseStarted ;
  NewConnectionPhaseEndedDelegate *onNewConnectionPhaseEnded ;
  NodeWasInfectedDelegate *onNodeWasInfected ;
  ThresholdExceededDelegate *onThresholdExceeded ;
  StabilityThresholdExceededDelegate *onStabilityThresholdExceeded ;
  ConnectionAttemptDelegate *onConnectionAttempt ;
  SocketAddedDelegate *onSocketAdded ;
  SocketDeletedDelegate *onSocketDeleted ;
};
```

### 3.2.3 Command Line Interface

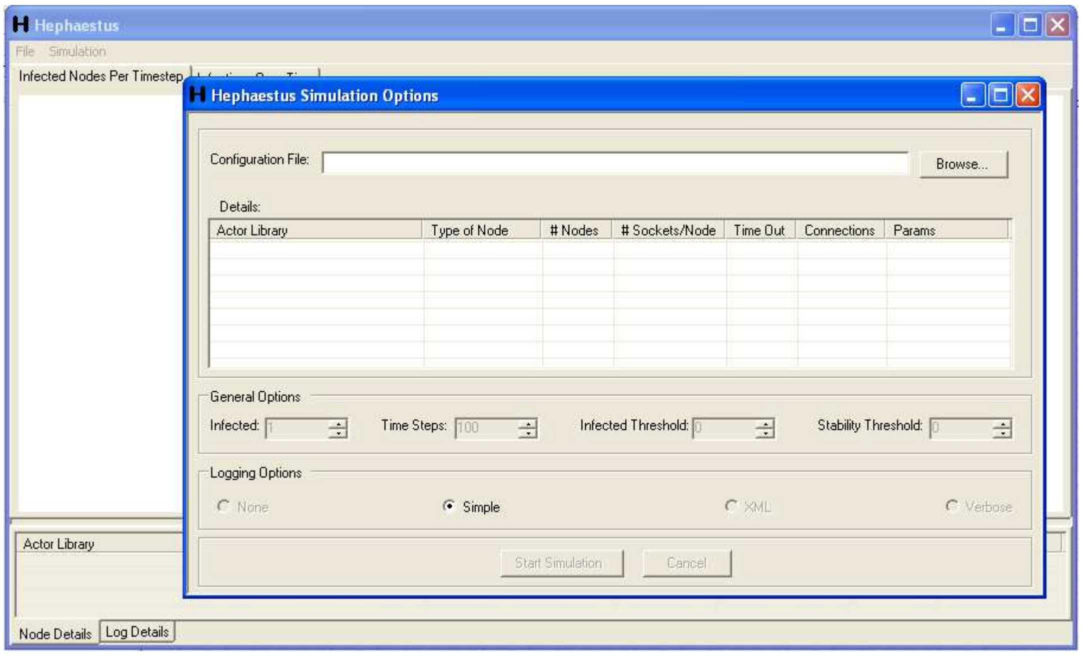
The Command Line Interface(CLI) is a console-based front-end to Hephaestus. This component has the same functionality as the Win32 User Interface with the exception of the graphing reports. Users can also specify several options to configure the simulation's execution. The CLI takes several parameters which allow the user to configure exactly how they want the simulation to be executed. They are listed below, with the required parameters specified in **bold** text.

- /**infected** - The number of nodes infected before the start of the simulation. Typically this should be set to 1.
- /**configfile** - The full path to the configuration file to use to set up the simulation.

- **/time** - The number of timesteps to run the simulation for. This will potentially be cut short as the options specified for /threshold and /stability take precedence over this option. In other words, this is the maximum number of timesteps the simulation will execute through.
- /threshold - The simulation will stop immediately when the total number of infected nodes reaches this value. This will limit the length of time the experiment will run after all the required nodes are infected.
- /stability - The simulation will stop after this number of timesteps has passed with no new infections. This will limit the length of time the experiment will run after all the nodes that will ever be infected are infected.
- /outputtype - The type of output to produce. If this option is not specified, no output will be given. Possible values are "xml", "verbose", and "simple".
  - The *xml* output generates valid XML output describing the execution steps of the simulation
  - The *verbose* output generates plain English descriptions of events that took place during the simulation.
  - The *simple* output generates 1 line for each timestep, with comma-separated fields. The first is the timestep number, starting with 1. The second field is the total number of infected nodes so far. The third field is the number of nodes infected during that timestep.
- /output - The file to write the output to in the specified output type. If this option is not specified, the output will be presented on the screen only.

### 3.2.4 Win32 User Interface

The Win32 User Interface of Hephaestus is designed to make the execution and viewing of experiment results as simple and user friendly as possible while still allowing for the flexibility and extensibility provided by the Hephaestus Command Line Interface. On launching the Win32 User Interface users will see a screen like the one in figure 3.2.



Configuration File:

Details:

| Actor Library | Type of Node | # Nodes | # Sockets/Node | Time Out | Connections | Params |
|---|---|---|---|---|---|---|

General Options

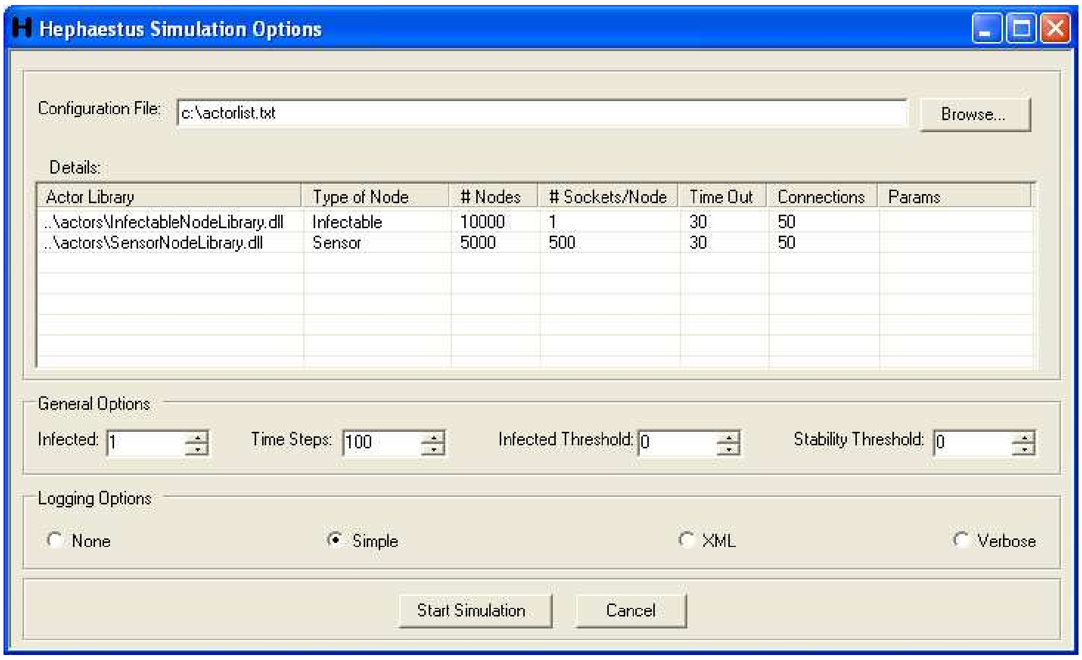
Infected: 1 Time Steps: 100 Infected Threshold: 0 Stability Threshold: 0

Logging Options

None [✓ Simple] XML Verbose

Figure 3.2: Initial Screen When the Win32 User Interface is Launched

Users should click on the “Browse...” button to locate a configuration file. Once they have selected a valid configuration file, the details of the file will be filled in to the “Details” list view and all the options will be enabled. It should look like the screen in figure 3.3, except the details area will likely contain different information.


Hephaestus Simulation Options

Configuration File: c:\actorlist.txt

Details:

| Actor Library | Type of Node | # Nodes | # Sockets/Node | Time Out | Connections | Params |
|---|---|---|---|---|---|---|
| ..\actors\InfectableNodeLibrary.dll | Infectable | 10000 | 1 | 30 | 50 | |
| ..\actors\SensorNodeLibrary.dll | Sensor | 5000 | 500 | 30 | 50 | |

General Options

Infected: 1 Time Steps: 100 Infected Threshold: 0 Stability Threshold: 0

Logging Options

None [✓ Simple] XML Verbose

Figure 3.3: The Win32 User Interface After a Configuration File Has Been Selected

At this point users can specify all the options they want to for this simulation execution. By double-clicking on one of the values in the details list, an edit box will appear and users can change the values that were read from the configuration file. In the "General Options" 4 options can be specified:

- *Infected* - The number of nodes infected before the start of the simulation. Typically this should be set to 1 unless a hit list of more than a single machine is being used.

- *Time Steps* - The number of timesteps to run the simulation for. This will potentially be cut short as the options specified for the Infected Threshold and Stability Threshold take precedence over this option. In other words, this is the maximum number of timesteps for the simulation to execute through.

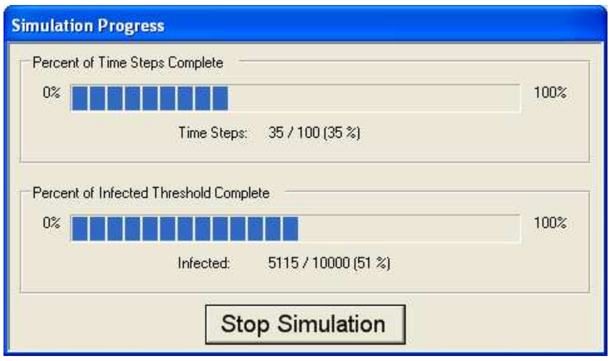
- *Infected Threshold* - The simulation will stop immediately when the total number of infected nodes reaches this value. This will limit the length of time the experiment will run after all the required nodes are infected. This also helps limit the X-axis of the graph reports to a meaningful range.

- *Stability Threshold* - The simulation will stop after this number of timesteps has passed with no new infections. This will limit the length of time the experiment will run after all the nodes that will ever be infected are infected. This also helps limit the x-axis of the graph reports to a meaningful range.

Also, in the "Logging Options" section users can specify a format for logging. These are the same options and the same format as the output types in the CLI. They are listed below.

- The *None* option generates no log output, the graphs will still be generated.
- The *XML* option generates valid XML output describing the execution steps of the simulation
- The *Verbose* option generates plain English descriptions of what took place during the simulation.
- The *Simple* option generates 1 line for each timestep, with comma-separated fields. The first is the timestep number, starting with 1. The second field is the total number of infected nodes so far. The third field is the number of nodes infected during that timestep.

After all necessary options have been specified all, users should click on the "Start Simulation" button. As the text implies, this will start the simulation. They will see a progress dialog showing the progress of the simulation. A sample screen is shown in figure 3.4.

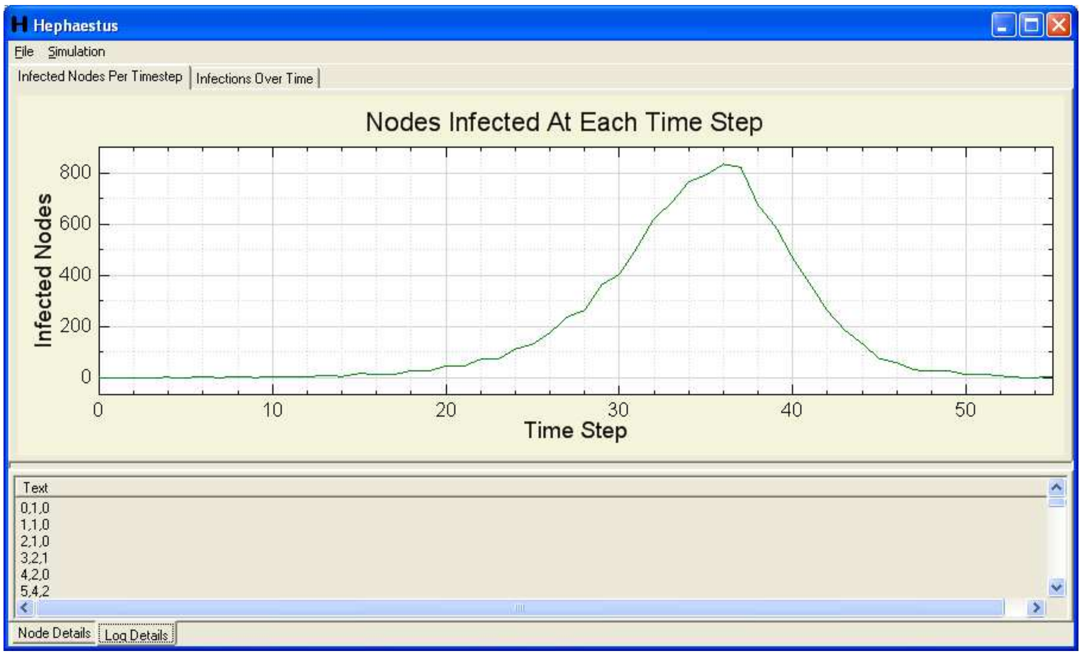

Simulation Progress
Percent of Time Steps Complete
0%
100%
Percent of Infected Threshold Complete
0%
100%
Stop Simulation

Time Steps: 35 / 100 (35 %)

Infected: 5115 / 10000 (51 %)

Figure 3.4: The Win32 User Interface Progress Window While Running a Simulation

The first progress bar is the percentage of the total time steps specified that have been executed. The second progress bar is the percentage of infected nodes versus the infected threshold specified in the options dialog. If the user did not specify an infected threshold this will always be at zero percent. When the simulation is complete, the results window will be displayed, which is shown in figure 3.5.



```
0,1,0
1,1,0
2,1,0
3,2,1
4,2,0
5,4,2
```

Figure 3.5: The Win32 User Interface Results Window

On the top of the Results Window, there are two tabs titled "Infected Nodes Per Timestep" and "Infections Over Time". The "Infected Nodes Per Timestep" graphs shows the number of nodes infected at each timestep. The "Infections Over Time" graph shows the total number of infections accumulated at each timestep. On the bottom of the screen there are also two tabs - "Node Details" and "Log Details". The "Node Details" tab will show the same options that were specified in the Details list view in the Options dialog. The "Log Details" tab will show a log of the simulation's results in the specified log format specified in the Options dialog.

There are also several menu items to be aware of:

- *File menu*
  - *Clear Log Window* - Clears the contents of the "Log Details" tab in the Results window
  - *Save Log Window Contents* - Saves the contents of the "Log Details" window to a file

  - *Save Current Chart* - Saves the currently selected graph to a Windows BMP file
  - *Exit* - Exits Hephaestus

- *Simulation menu*
  - *Start...* - Opens the Options dialog so the user can change the options and start a new simulation run.

### 3.2.5 ScPl

ScPl is a plotting library for Microsoft's .NET platform. It is used in the Win32 User Interface to generate the graphs seen after a simulation execution. It is free provided that their license agreement and copyright notices are included in the distribution. This has been done for Hephaestus, and it is included here again. It works very well and is extremely simple to use in code.

```
ScPl - A plotting library for .NET
Copyright (C) 2003, the ScPl team.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the following text in
the documentation and / or other materials provided with the
distribution:

"This product includes software developed as part of
the ScPl plotting library project available from:
http://www.netcontrols.org/scpl/ "

------------------------------------------------------------------------

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

### 3.2.6 Actor Libraries

Actor libraries are Windows DLLs that plug in to the simulator engine to represent different types of nodes in the simulator. Five different actor libraries have been provided to use "out of the box", but users can easily create their own for further extensibility, as is described in section 3.3.2. The provided actor libraries, and how they differ from each other are described below:

- *IncreasingConnectionTimeoutNodeLibrary.dll* - Contains the "Increasing Connection Timeout Node" node type. Every time that an infected node attempts to connect to this node, the infecting node will have to wait longer before the connection is accepted. This value is specified as a parameter to the node type in the configuration file. So if this value is set to 5, the first connection attempt will not be answered for 5 timesteps. The next one will not be answered for 10 timesteps, the third for 15, etc. This node will never be infected. Since it can never be infected it will never try to infect other nodes.

- *InfectableNodeLibrary.dll* - Contains the "Infectable" node type. This node will always accept all connection attempts and infection attempts by any infected node. Once infected it will always attempt to connect and infect another node.

- *LimitedConnectionsNodeLibrary.dll* - Contains the "Limited Connections Node" node type. This node will always accept connection attempts, will always accept infection attempts, but will only try to connect and infect other nodes for a variable number of times. This value is specified as a parameter to the node type in the configuration file. So if the value is specified to be 10, this node will only try to connect to and infect 10 other nodes ever in its lifetime.

- *SensorNodeLibrary.dll* - Contains the "Sensor" node type. This node will never accept connection attempts and will never accept infection attempts. Since it can never be infected it will never try to infect other nodes.

- *UninfectableNodeLibrary.dll* - Contains the "UnInfectable" node type. This node will always accept connection attempts from other nodes but will always deny infection attempts. Since it can never be infected it will never try to infect other nodes.

## 3.3 How Hephaestus Can Be Enhanced and Extended

There are 2 major ways to extend and enhance the simulator. Adding new actor libraries allows the user to see how a distribution of a particular type of infective or preventative measure affects the spread of malicious mobile code.

Using configuration files allows the user to configure the parameters of each actor library to be used in a simulation.

### 3.3.1 Configuration Files

Configuration files in Hephaestus are the main input to the simulator. It contains the information about the nodes that will be simulated in a given execution of Hephaestus. This information includes the following:

- The path to the actor library containing the actor definition
- The number of nodes created from that actor library
- The number of sockets each node has
- The number of timesteps that a connection attempt to a node of that type will timeout in
- The number of connection attempts a node of that type will try before moving on to another machine
- Any additional parameters specific to that node type. This will vary between actor libraries.

A configuration file contains at least 1 node set definition, and optionally comment lines as well. Each node set definition is specified on its own line, and each line's fields are comma delimited. Here is a sample configuration file:

```
; This is a comment line
..\actors\InfectableNodeLibrary.dll,10000,1,30,50
..\actors\SensorNodeLibrary.dll,5000,500,30,50
```

This file specifies that there will be two types of actors involved in this execution of Hephaestus. The first is the node type from InfectableNodeLibrary.dll. There will be 10,000 nodes of that type. Each one will have 1 socket, will allow 30 seconds before connection attempts are timed out, and will have 50 connection retries before giving up. The second is the node type from SensorNodeLibrary.dll. There will be 5,000 nodes of that type. Each one will have 500 sockets, will allow 30 seconds before timing out a connection attempt, and will have 50 connection retries before giving up. Neither of these lines contains parameters because neither node type allows any parameters.

### 3.3.2 Adding Actor Libraries

Each actor library used by the simulator needs to inherit from and implement abstract methods of 2 classes. Each actor library is a Windows DLL and once compiled, can be used by the simulator with no extra effort. In effect, they are plugins to the simulator. The classes and methods that need to be implemented are documented below. Also listed are two enumerations that are useful to know what possible return values are for some methods in those classes. All use the namespace "SimulatorEngine". The complete source code for a sample actor library can be found in Appendix B. The enumerations are below:

```
//**************************************************************************
// possible connection attempt results
//**************************************************************************
enum ConnectionResult
{
    Success = 0,            // the connection was successful
    Denied,                 // the connection attempt was denied
    NoAnswer                // the node did not respond to the connection attempt
};

//**************************************************************************
// possible infection attempt results
//**************************************************************************
enum InfectionResult
{
    NoPreviousAttempt = 0,  // this node has never had an infection attempt against
                            // it
    Infected,               // the node was infected
    NotInfected,            // the infection attempt was denied
    NoInfectionAnswer       // the node did not respond to the infection attempt
};
```

Here is the one method that needs to be implemented in the subclass of *class NodeLibrary*:

```
//**************************************************************************
// Method: CreateNode
// Description: Creates a new NetworkNode object and returns it
//
// Parameters:
// params -        a list of parameter strings to pass to the node
// numSockets -    the number of sockets for the new node
// timeOut -       the connection timeout for the new node
// numConnections - the number of connection attempts for the new node
//
// Return Value: a new network node object
//**************************************************************************
NetworkNode *CreateNode(vector<string>*params, unsigned short numSockets,
    unsigned short timeOut, unsigned char numConnections);
```

Below are the methods to implement in the class subclassed from *class NetworkNode*:

```
//**************************************************************************
// Method: NetworkNode
// Description: Constructor for the NetworkNode class
//
// Parameters:
// params - a list of parameter strings. Can be anything. These
// are passed in directly from the configuration files
//
// Return Value: None
//**********************************************************************
NetworkNode(vector<string> *params);

//**********************************************************************
// Method: WillAttemptToConnectToNode
// Description: returns whether or not this node will attempt to connect
// to the specified node
//
// Parameters:
// nodeToConnectTo - the node to determine whether or not to connect to
//
// Return Value: true if it should attempt a connection, false otherwise
//**********************************************************************
bool WillAttemptToConnectToNode(NetworkNode *nodeToConnectTo);

//**********************************************************************
// Method: ConnectToNode
// Description: tries to connect to this node from another node
//
// Parameters:
// connectingNode -                the node trying to connect
// nodeToConnect -                the node being connected to
// timeSocketHasBeenOpen - the number of timesteps the socket has been open
// between the 2
//
// Return Value: a connection result enumeration value
//**********************************************************************
ConnectionResult ConnectToNode(NetworkNode *connectingNode, NetworkNode *nodeToConnect,
    unsigned short timeSocketHasBeenOpen);
//**********************************************************************
// Method: AttemptToBeInfected
// Description: called when the infectingNode is trying to infect this node
//
// Parameters:
// infectingNode - the node trying to infect this one
//
// Return Value: a infection result enumeration
//**********************************************************************
InfectionResult AttemptToBeInfected(NetworkNode *infectingNode);
```

## 3.4 A Simple Walk-Through Tutorial of a Hephaestus Experiment

In order to help users understand how to use Hephaestus and to better understand its results, this section will walk the reader through a simple experiment. It will let the user take a look to see what happens to the spread of an infection if software is installed that allows infections to occur freely, but then limits the number of times that node can try to infect other nodes on all nodes in the network. The user will observe results when the number of infection attempts is unlimited, and also when it is limited to 10, 5, 3, and 2 attempts per infected node. We will also assume that 40% of the nodes on our network are uninfectable due to correct service packs being applied, wrong operating system, etc.

First we will run the simulation with no limitation on the infection attempts. The first thing to do in order to run this experiment is to create a configuration file set up the way we want it. For the simulation with no limitations on the infection attempts we will need two node types, both of which are provided with the Hephaestus distribution. They are in the actor libraries "InfectableNodeLibrary.dll" and "UnInfectableNodeLibrary.dll" and can be found in the "actors" folder of the distribution. Here is the text of the configuration file to use when the configuration file is put in the root folder of the distribution:

```
.\actors\InfectableNodeLibrary.dll,60000,1,30,50
.\actors\UnInfectableNodeLibrary.dll,40000,1,30,50
```

Figures 3.6 and 3.7 are sample screenshots of this simulation when run through the Hephaestus Win32 User Interface. All 60,000 potentially infected nodes were infected rather quickly.

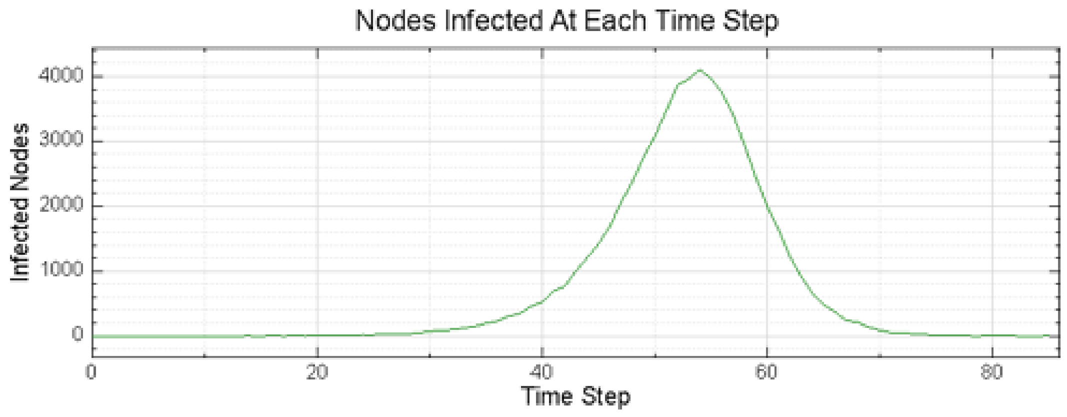


Figure 3.6: Nodes infected at each timestep with no connection limitation

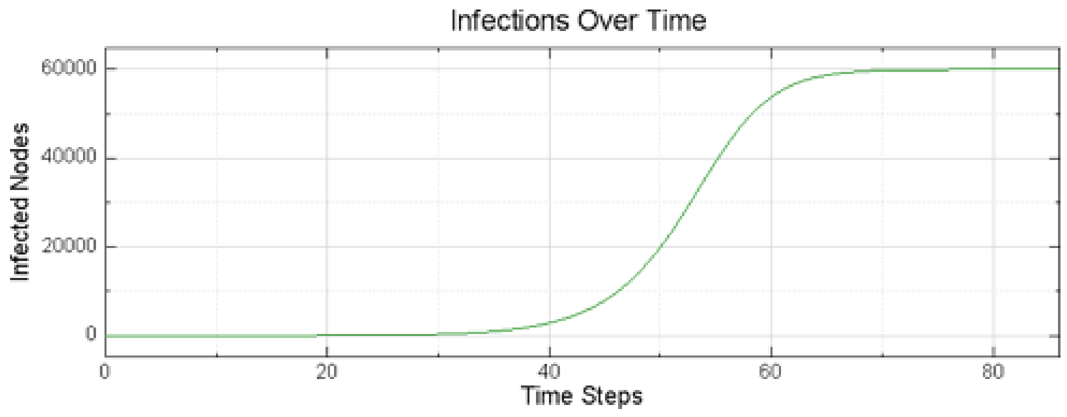


Figure 3.7: Nodes infected over time with no connection limitation

Now the user should create a new actor library to meet our needs. For simplicitly, this walk-through uses the LimitedConnectionsNode actor. The complete source code for this actor type can be found in Appendix B. Once the user has created and compiled this library, the simulator engine will take care of all the interactions with the actor, we just need a configuration file to run the tests limiting the connection attempts to 10, 5, 3, and 2. The new configuration file is listed below:

```
.\actors\LimitedConnectionsNodeLibrary.dll,60000,1,30,50,10
.\actors\UnInfectableNodeLibrary.dll,40000,1,30,50
```

Notice the new library name on the first line of the configuration file as well as the new value of “10” added to the end of the line. That makes the string “10” (which will be converted to an integer in the constructor) the first parameter passed to the constructor of the library. The results of running this configuration file using Hephaestus are shown in figures 3.8 and 3.9. Less than 56,000 nodes were infected so limiting connections is resulting in a slowdown in the spread of the infection.

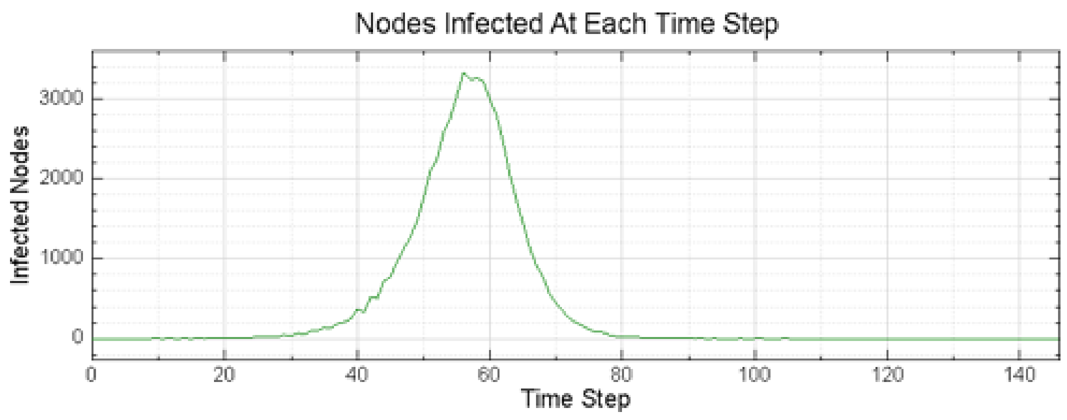


Figure 3.8: Nodes infected at each timestep with a connection limit of 10

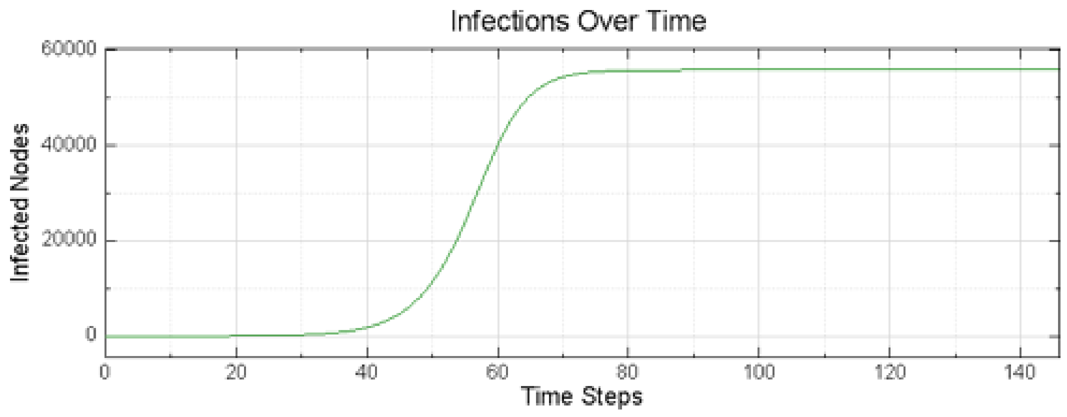


Figure 3.9: Nodes infected over time with a connection limit of 10

To run the experiment with connections limited to 5, all that is needed is to change the parameter passed on the first line of the configuration file. So instead of “10” at the end of the first line, we change the value to “5” and run Hephaestus on this configuration file again. The results are shown in figures 3.10 and 3.11, and this time just over 50,000 nodes were infected.

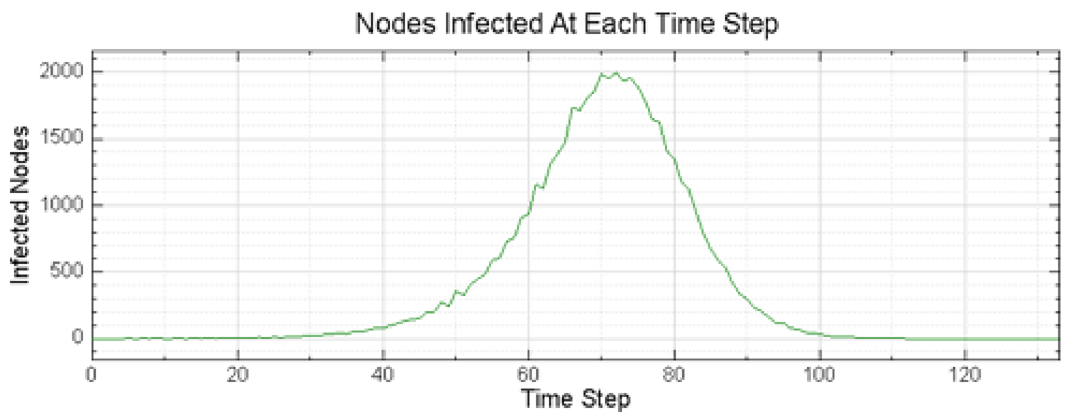


Figure 3.10: Nodes infected at each timestep with a connection limit of 5

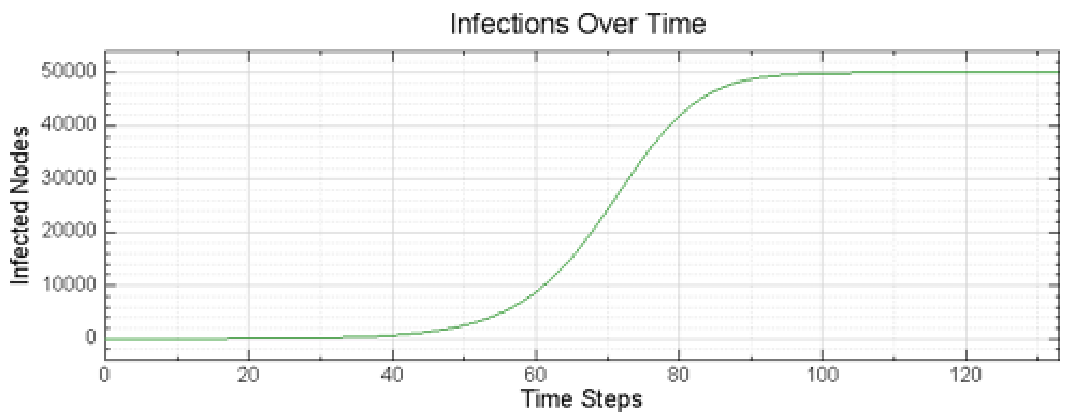


Figure 3.11: Nodes infected over time with a connection limit of 5

Results for a connection limit of 3 are shown in figures 3.12 and 3.13, while results for a connection limit of 2 are shown in 3.14 and 3.15. Just under 20,000 nodes were infected for a limit of 3, while just 5 out of the 60,000 nodes were infected with a connection limit of 2.

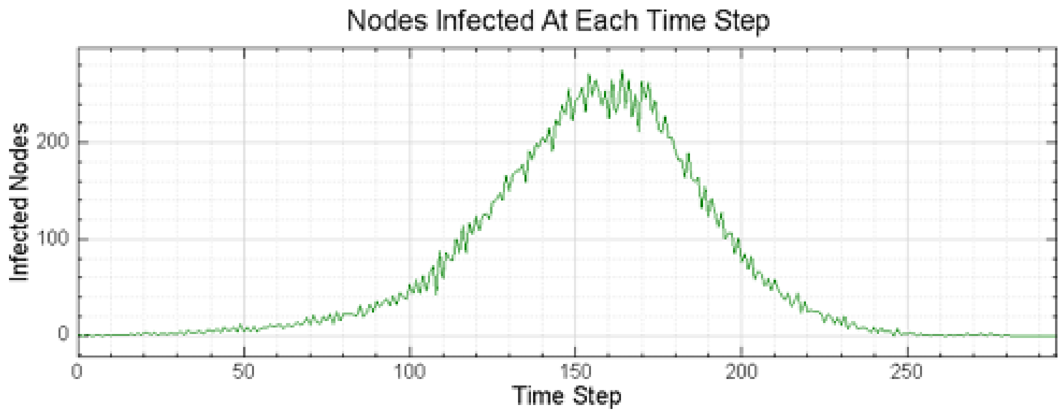


Figure 3.12: Nodes infected at each timestep with a connection limit of 3

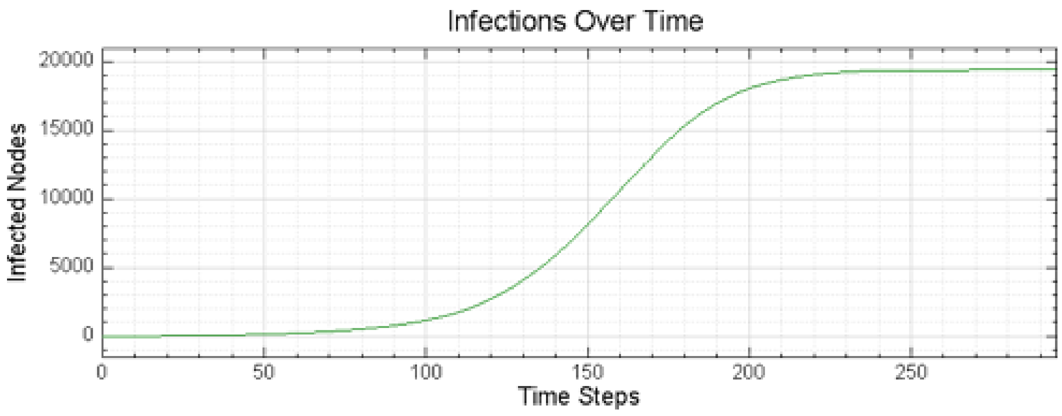


Figure 3.13: Nodes infected over time with a connection limit of 3

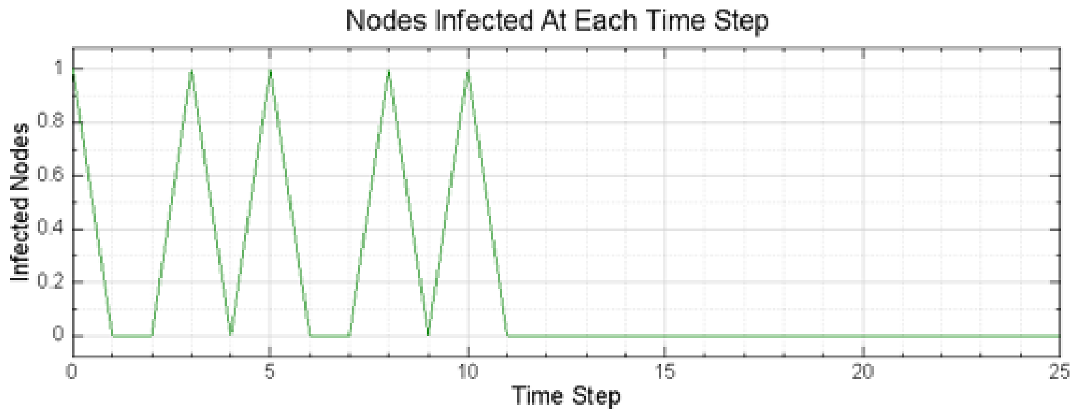


Figure 3.14: Nodes infected at each timestep with a connection limit of 2

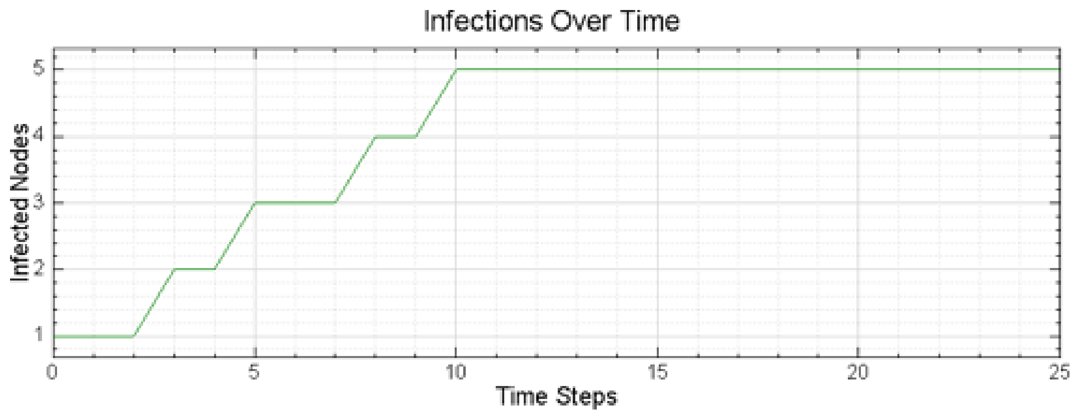


Figure 3.15: Nodes infected over time with a connection limit of 2

The observations that can be taken away from this experiment are that if the user had some software installed on potentially infected machines, and configured the software to only allow outgoing infections a limited number of times, the software can actually slow or even stop the spread of that malicious code even without signature-based anti-virus software installed. Of course, without the anti-virus software many machines will be damaged, but the point is the spread can be stopped even without it.

# Chapter 4

# Experiments Using Hephaestus

The flexibility and extensibility of Hephaestus allows a wide range of experiments to be run to determine the effectiveness of various simulated spread prevention techniques. By studying the results of those experiments, hopefully more effective real-world virus prevention techniques can be developed and applied to real computers and networks. This chapter presents the implementations and results of four such experiments: one with no prevention measures, Monoculture, Lossy Detection, and Tarpits.

## 4.1 No Prevention Measures

The first experiment that was run to make sure the simulator was working as expected was to simulate a virus that spreads without prevention measures. To run this experiment we only needed 2 actor types - Infectable and Uninfectable. Five experiments were then run under Hephaestus with different ratios of infectable to uninfectable nodes, each with a total of 100,000 nodes.

The first experiment was with 100,000 infectable nodes and zero uninfectable nodes. In essence this simulation is approximately the expected result if you ran the Weaver simulator[19] with 100,000 nodes with each timestep being 1 second in Weaver's simulator and a hitlist size of 0. As simple, although certainly not complete, verification we can compare the curve produced by Hephaestus to the random curve shown by Weaver[19] to ensure that they agree. The results from Hephaestus are shown in figures 4.1 and 4.2. The results from Weaver's experiments are shown in figure 4.3. The configuration file used for the simulation is listed below. The configuration files used for the other simulations in this section are identical other than the number of nodes of each actor type.

```
..\actors\InfectableNodeLibrary.dll,100000,1,30,50
..\actors\UninfectableNodeLibrary.dll,0,1,30,50
```

The next simulation run was one with three quarters of the population infectable. We expected the curve to have the same shape as with 100 percent infectable, but take more timesteps to complete. As shown in figures 4.4 and 4.5, this is in fact what the simulation showed.

Next a 50:50 ratio of infectable to uninfectable nodes was simulated. Again, we expected the curve to have the same shape as with 100 and 75 percent infectable, but take more timesteps to complete. Again, that was the case (Figures 4.6 and 4.7)

Another simulation was run with one quarter of the nodes being infectable. Results were again consistent with the expected results (Figures 4.8 and 4.9)

Finally, a simulation with just one percent of the nodes being infectable was run. The results are shown in figures 4.10 and 4.11. This also agreed with our expectations of the curve, but there was an added verification in these results. Notice the "jaggedness" of the graphs in figures 4.9 and 4.11. The reason this jaggedness occurred is due to the fact that at these ratios there was a high chance that many of the infected nodes would try to infect nodes that were already infected or uninfectable. The number of uninfectable or already infected nodes hit each timestep varied greatly and so it should not have been expected that the curve would be generally as smooth as those with a high ratio of infectable nodes.

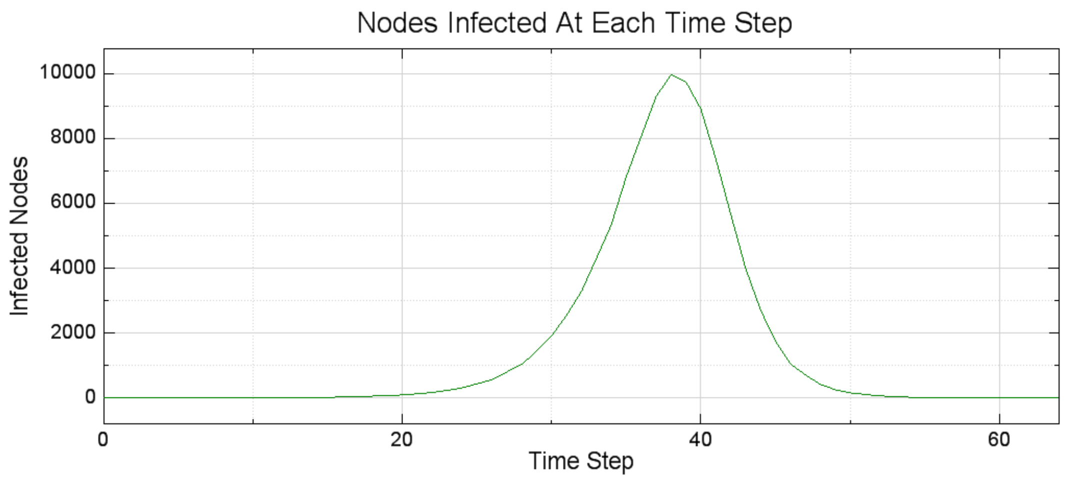


Figure 4.1: Infectable nodes infected at each timestep using Hephaestus where all 100,000 nodes are infectable

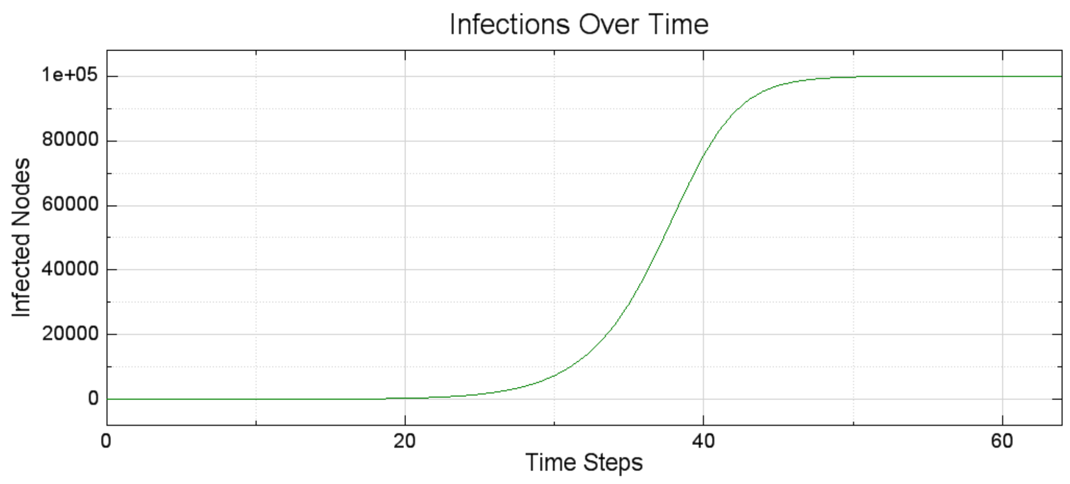


Figure 4.2: Infectable nodes infected over time using Hephaestus where all 100,000 nodes are infectable

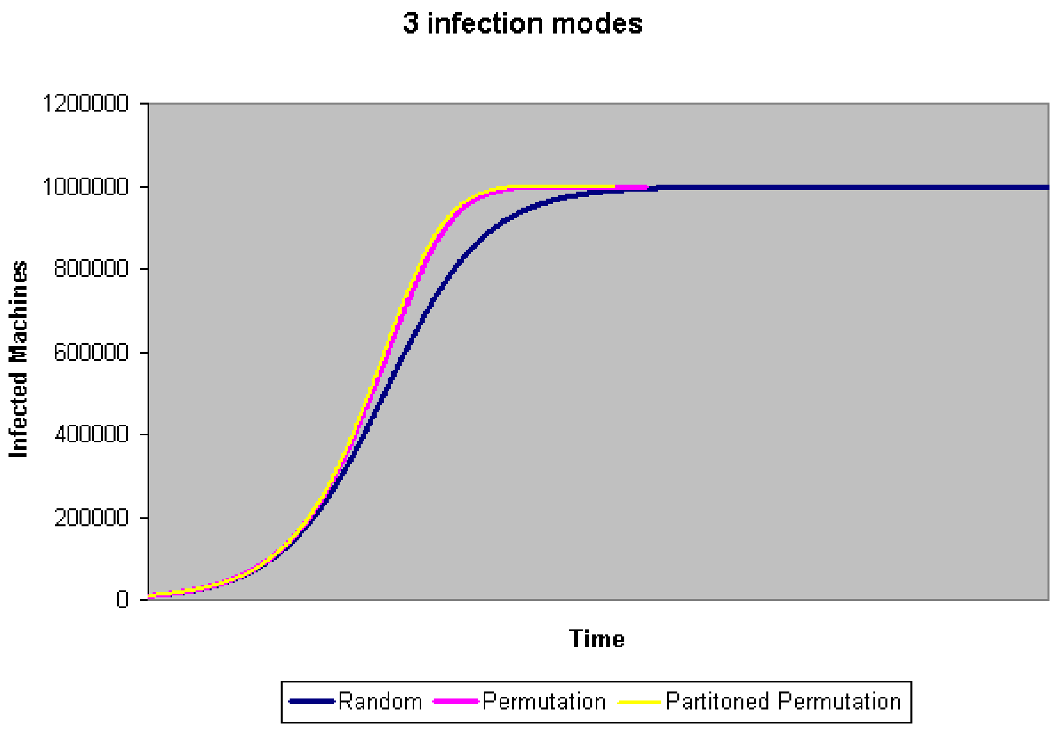


Figure 4.3: Results of the Weaver simulator

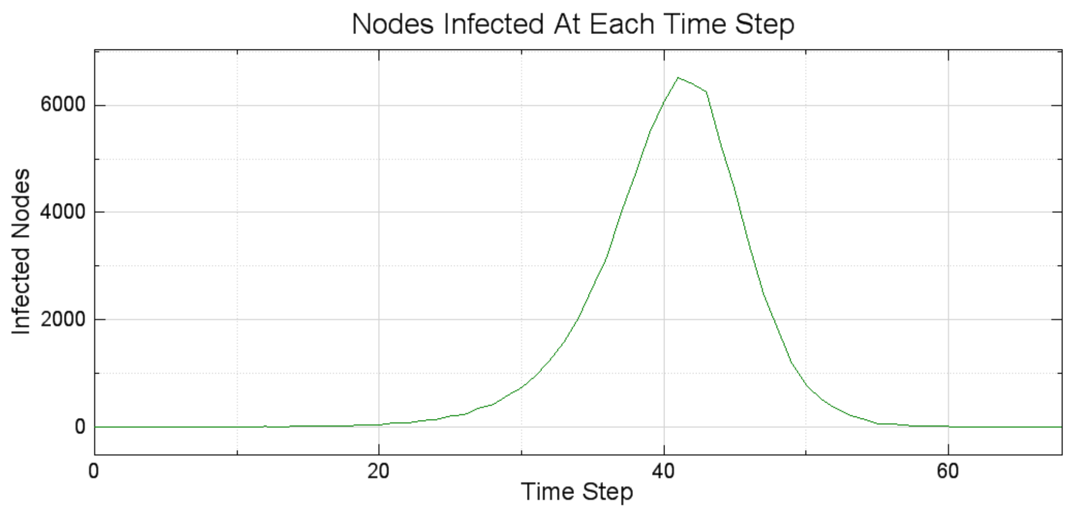


Figure 4.4: Infectable nodes infected at each timestep using Hephaestus with a 75,000 to 25,000 infectable vs uninfectable ratio

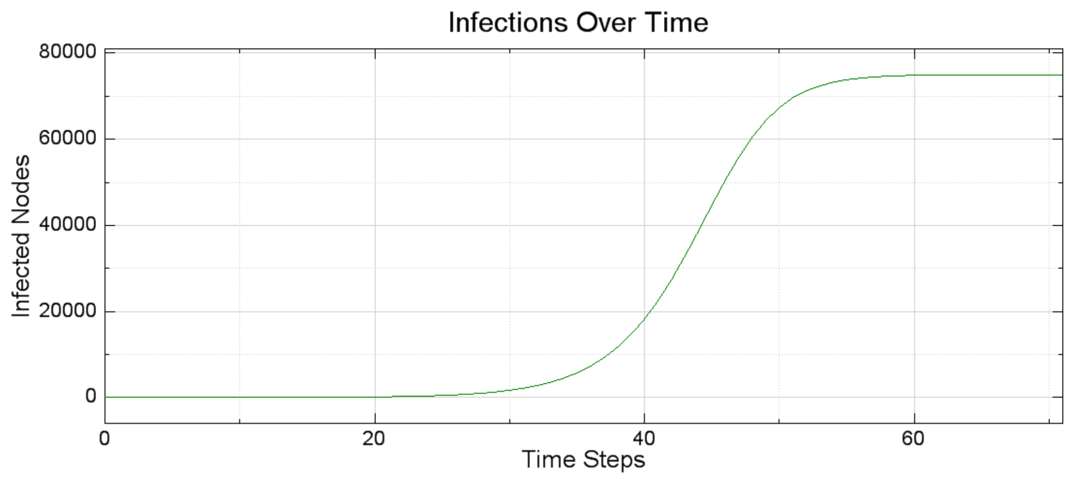


Figure 4.5: Infectable nodes infected over time using Hephaestus with a 75,000 to 25,000 infectable vs uninfectable ratio

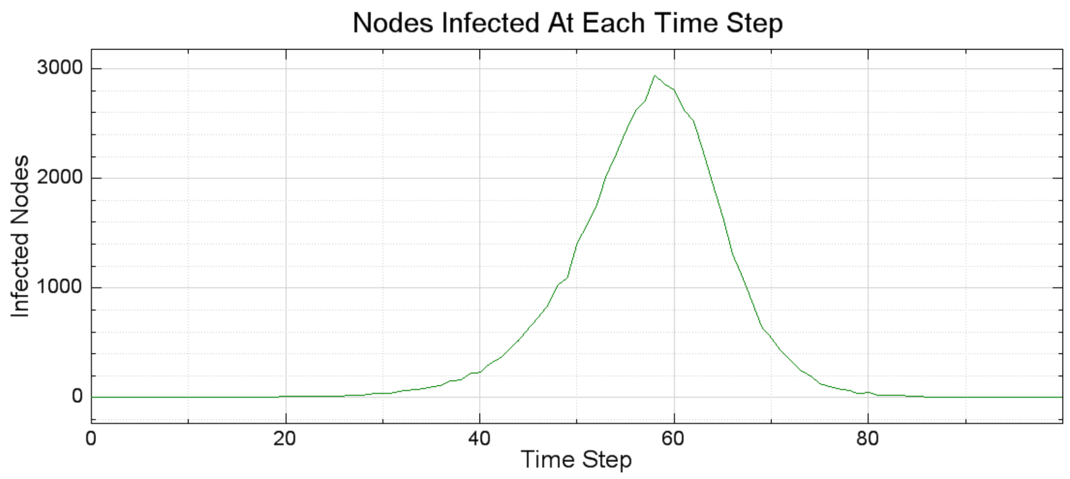


Figure 4.6: Infectable nodes infected at each timestep using Hephaestus with a 50,000 to 50,000 infectable vs uninfectable ratio

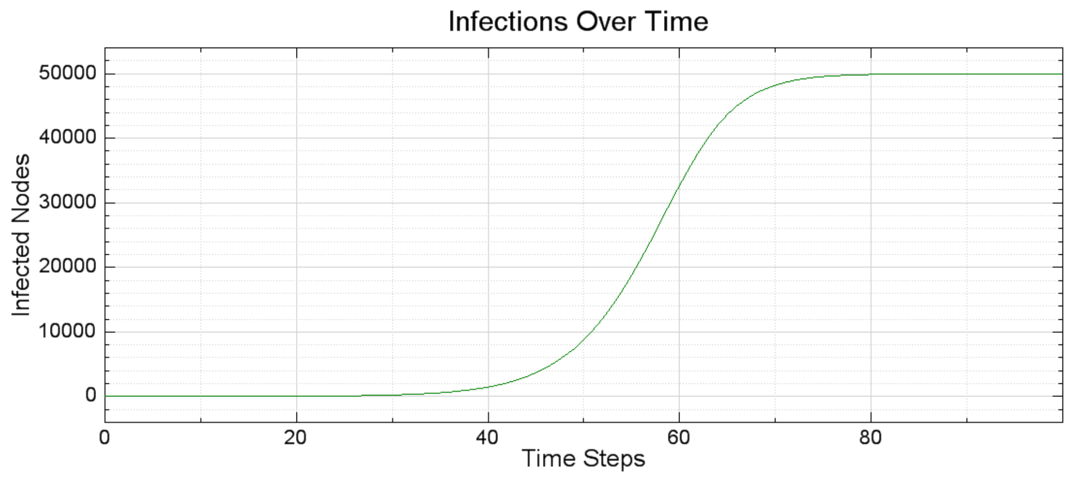


Figure 4.7: Infectable nodes infected over time using Hephaestus with a 50,000 to 50,000 infectable vs uninfectable ratio

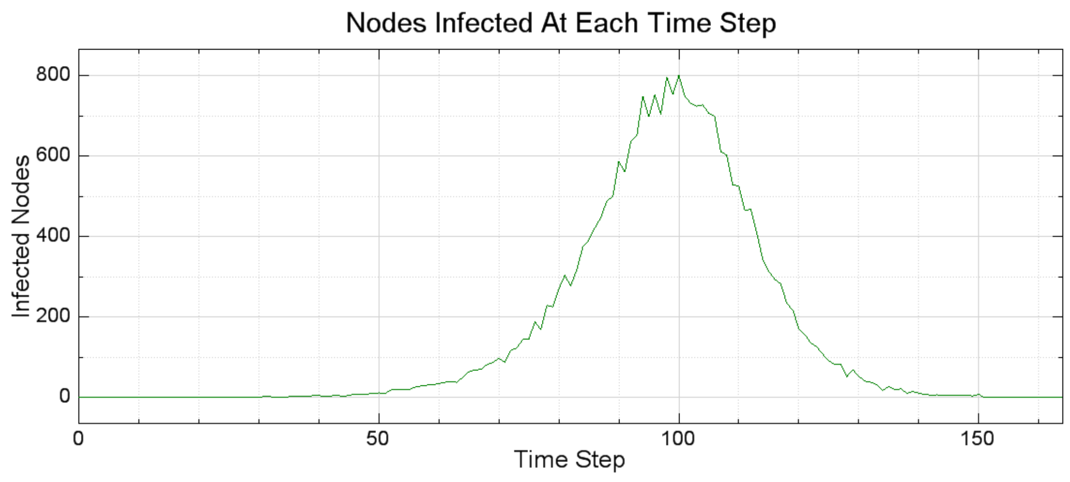


Figure 4.8: Infectable nodes infected at each timestep using Hephaestus with a 25,000 to 75,000 infectable vs uninfectable ratio

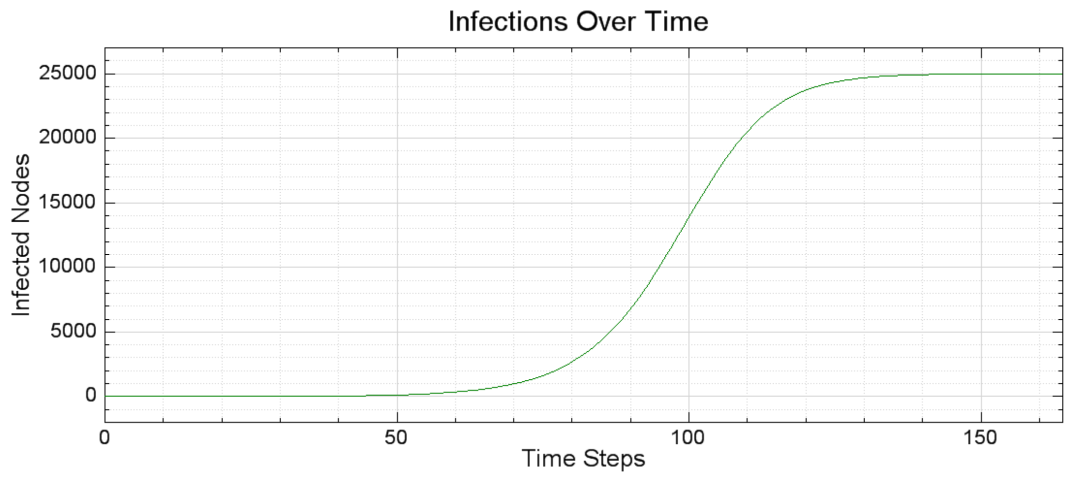


Figure 4.9: Infectable nodes infected over time using Hephaestus with a 25,000 to 75,000 infectable vs uninfectable ratio

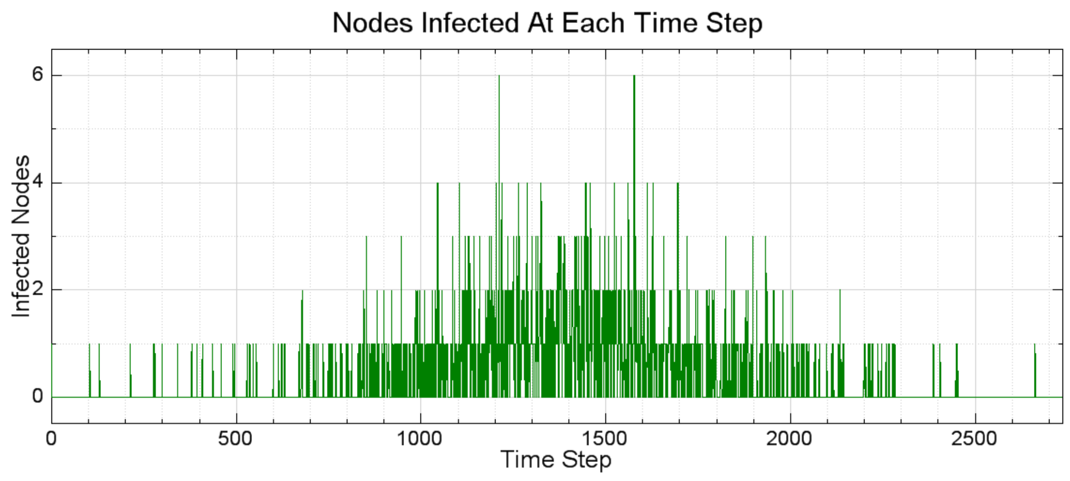


Figure 4.10: Infectable nodes infected at each timestep using Hephaestus with a 1,000 to 99,000 infectable vs uninfectable ratio

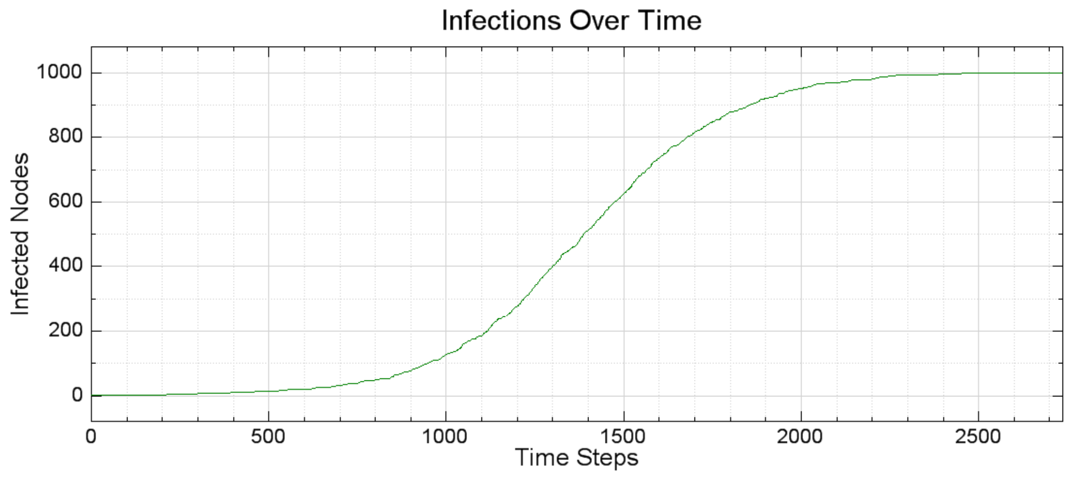


Figure 4.11: Infectable nodes infected over time using Hephaestus with a 1,000 to 99,000 infectable vs uninfectable ratio

## 4.2 Monoculture

In 2003, a paper published by several highly-respected cybersecurity experts stated that a monoculture in computer networks is not only undesirable, but also critically dangerous[25]. They also state that a monoculture currently exists due to most of the computers in the world running Microsoft software. Since that time, other respected experts have determined that the diversity needed to effectively slow down the spread of malicious mobile code including viruses to a level where humans can respond to the threat is nearly impossible to achieve in real-world networks[26]. Hephaestus can be used to approximate what level of diversity is needed to allow diversity alone to be a deterrent of virus spread. This section attempts to do just that.

There are two separate ways to set up this experiment. In one way, a constant number of total nodes can be used while the number of infectable and uninfectable nodes vary with each diversity change. For instance, the experiment could use a constant total node count of 100,000 nodes, and for 2.5 percent diversity there would be 97,500 infectable nodes and 2,500 uninfectable nodes. At 5 percent diversity, the ratio would be 95,000 to 5,000 respectively. In the other way, the number of infectable nodes would remain constant while the uninfectable count would change to make the diversity percentage correct. So if 95 percent is being modeled and 100,000 infectable nodes are chosen, there would be 1.9 million uninfectable nodes in the experiment. The first method was chosen because it is much easier to interpret the results since the total number of nodes is always constant.

For this experiment, we ran the Hephaestus CLI through a set of 40 configuration files 20 times each with 500 timesteps for a total of 800 output files, each in the "Simple" output format. Each configuration file had the basic format shown below:

```
.\actors\InfectableNodeLibrary.dll,NI,1,30,50
.\actors\UnInfectableNodeLibrary.dll,NU,30,50,10
```

In each configuration file there were 100,000 total nodes, and there were 40 configuration files since we varied the diversity between each configuration file from 2.5% to 100% in 2.5% increments. Each of these percentages will be identified by the variable $P$ in the equations below. $NI$ was the number of infectable nodes in the configuration file, and $NU$ was the number of uninfectable nodes. Therefore $NU$ is defined by the equation:

$$NU = 100000 * (P/100) \tag{4.1}$$

and $NI$ is defined by:

$$NI = 100000 - NU \tag{4.2}$$

The purpose of running each of the 40 configuration files through Hephaestus 20 times was to help average out the data due to the randomness of the scanning involved when an infected node attempts to infect a new node. After obtaining the 800 results files, they were all merged into a single three column tab delimited text file and run through Matlab 7 to produce the plot file. The resulting plot file for this experiment is shown in figure 4.12, but to avoid confusion it should be noted that the total number infected is 20 times that of a single run, as those numbers were summed, not averaged, between the runs. This had no effect on the meaning of the results, rather it only affected the scale of the results, as the shape of the graph would be identical had those numbers been averaged. This graph very closely agrees with Dr. Richard Ford's monoculture experiment created using a very early Linux-based proof of concept version of Hephaestus shown in figure 4.13[26].

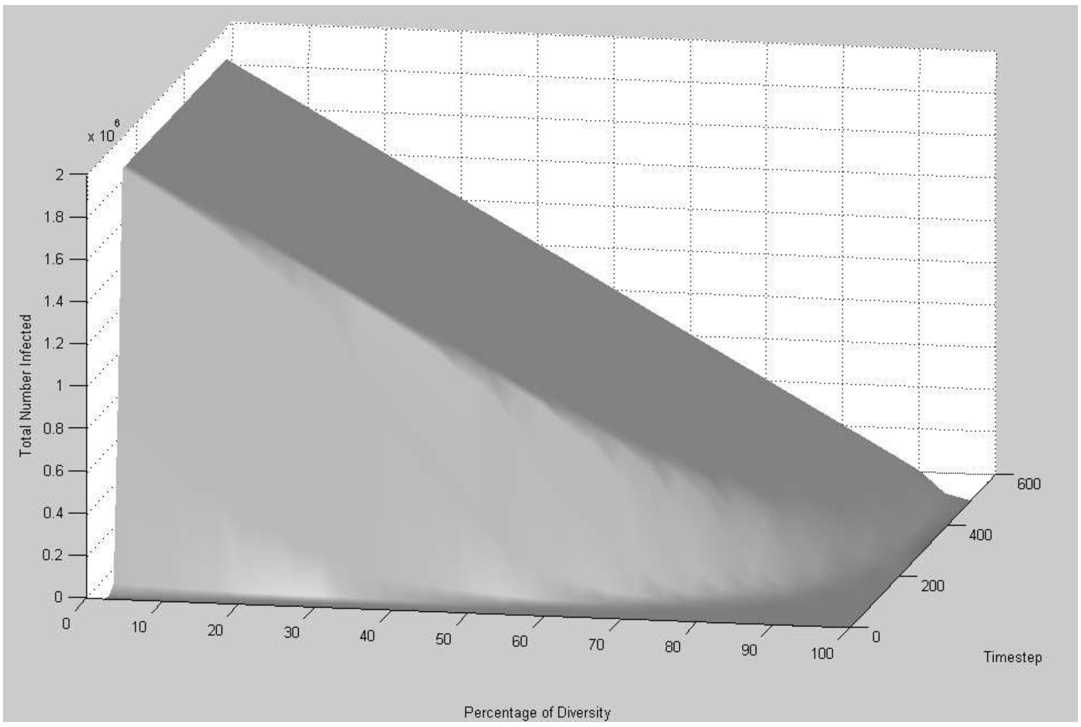


Figure 4.12: Monoculture plot for a constant 100,000 nodes under Hephaestus

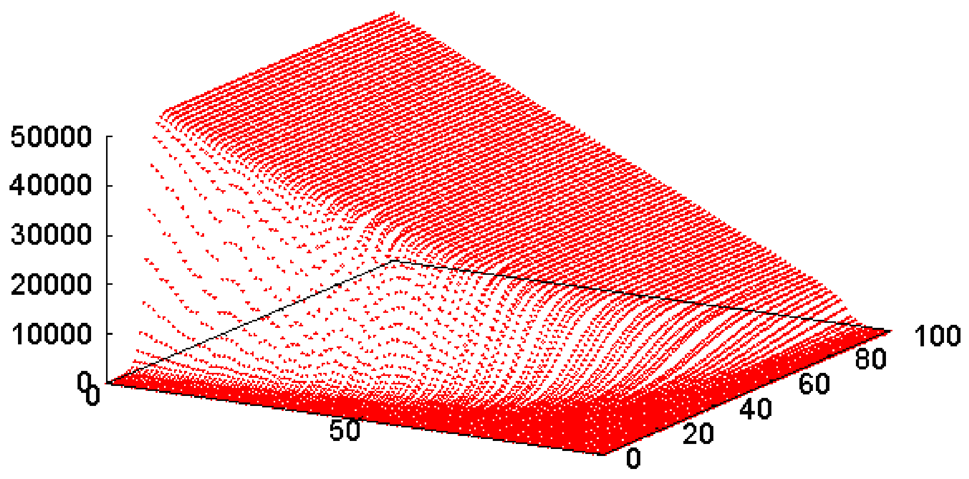


Figure 4.13: Monoculture plot from Dr. Richard Ford's Monoculture article

A plot to better show the slope in the graph resulting from the experiment is shown in figure 4.14. Even near 75% diversity, meaning only one out of every four nodes on the network is infectable, almost all infectable nodes were infected in under 100 timesteps. This is well under the amount of time that virus signatures can be created and distributed for quick spreading viruses and worms. An interesting result of this experiment is that the authors of [25] were correct in that having a diverse node population does in fact reduce the damage caused and the number of machines infected by a particular malicious application when compared to a monoculture. However, Dr Ford was also correct in [26] in that if the goal of avoiding monoculture is to prevent wide spread infection, it does not matter if there is a monoculture or not. By avoiding a monoculture, fewer machines will be affected by an outbreak because fewer machines will be vulnerable, but every vulnerable machine will still be quickly infected regardless of the diversity.

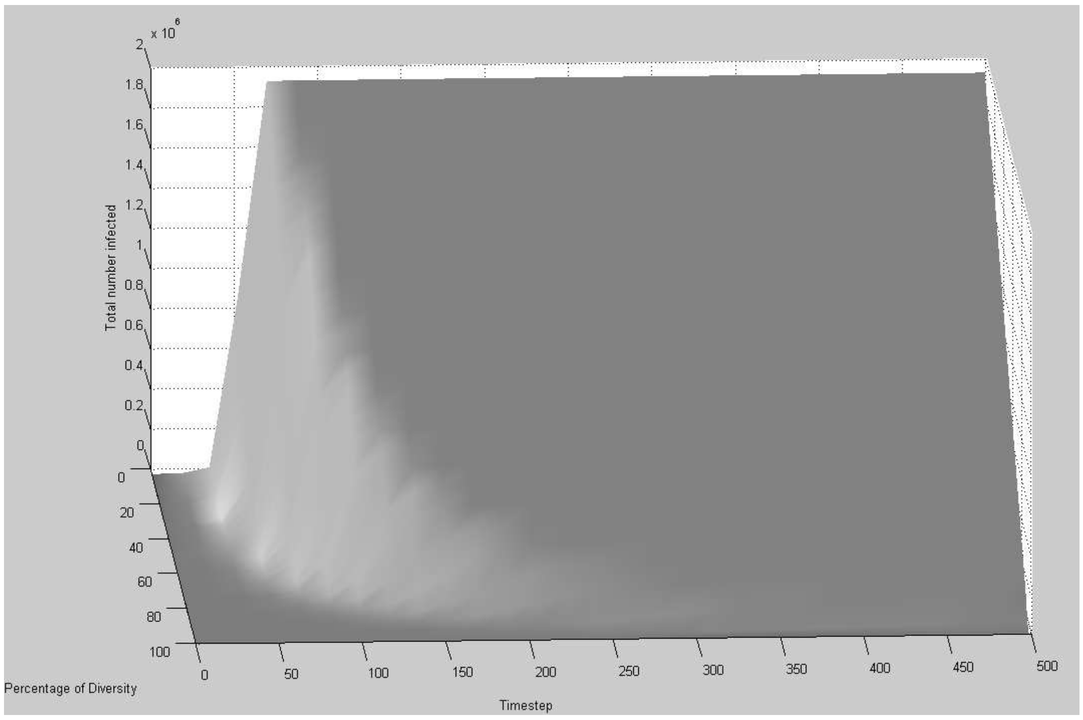


Figure 4.14: Face-on monoculture plot for a constant 100,000 nodes under Hephaestus

## 4.3 Lossy Detection

A simple potential method of slowing the spread of a self-replicating virus or worm takes advantage of the fact that other than the target IP address to be infected, a virus will send out the same set of packets over the network to infect multiple targets. If software on the infected computer looks for this identical outgoing traffic (other than the destination IP address of course), it can let a few sets of the infection attempts pass through and then at a certain threshold number can block that traffic until the user explicitly allows it to go through. In effect, this would act as an outgoing traffic firewall that allows a few connections to get through before dynamically preventing the traffic from future outgoing infection attempts. This technique will be called lossy detection throughout the remainder of this thesis.

There are two questions in this potential approach. First, will this method work at all? If some traffic gets out, will the lossy detection really slow the rate of spread to bring the virus spread below epidemic levels? The other question is if it *can* slow the rate of spread, what threshold levels *will* sufficiently slow down the spread rate of the infection? Both of these questions can be easily answered using Hephaestus.

### 4.3.1 No Suppression

First, we will show the spread of an infection when there is no suppression of outgoing infection attempts. This will make it easier to understand the difference in spread that adding limits to the outgoing attempts has. Figure 4.15 contains the graph of a typical infection with no suppression. The configurations files used to run the experiments look like the one below:

```
.\actors\InfectableNodeLibrary.dll,NI,1,30,50
.\actors\UnInfectableNodeLibrary.dll,NU,30,50,10
```

where *NI* was always a constant 100,000 nodes and $NU$, the number of uninfectable nodes, is defined by the equation

$$NU = (100000/(P/100)) - 100000 \tag{4.3}$$

There were 40 configuration files since we varied the diversity $P$ between each configuration file from 2.5% to 100% in 2.5% increments.

As expected, 100 percent of the nodes were infected quickly even at low infectable percentages. Next we will show the effects of suppressing the outgoing infection attempts.

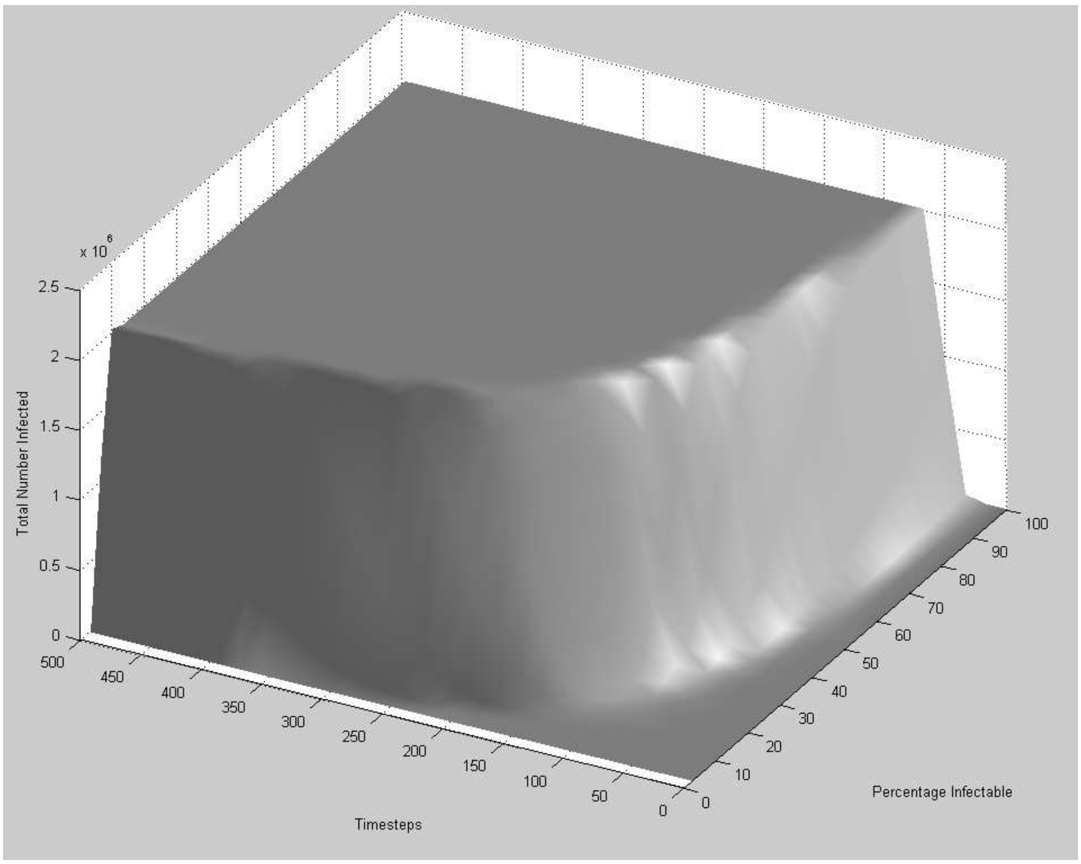


Figure 4.15: Graph of virus spread with no outgoing infection attempt suppression

### 4.3.2 Ten Outgoing Infections Per Node

The experiment for limiting outgoing connection attempts to 10 was run exactly like the one with no suppression, but the configuration files were slightly different. They changed to look like the one below:

```
.\actors\LimitedConnectionsNodeLibrary.dll,NI,1,30,50,10
.\actors\UnInfectableNodeLibrary.dll,NU,1,50,10
```

where $NI$ was always a constant 100,000 nodes and $NU$, the number of uninfectable nodes, is defined by the equation

$$NU = (100000/(P/100)) - 100000 \qquad (4.4)$$

Running this experiment generated the graph shown in figure 4.16. Clearly this shows that fewer infections occurred early in the experiment than with no suppression (figure 4.15), but it is hard to quantify the difference by visually comparing the graphs. So we generated a graph that was the difference in the number infected between the two graphs and that graph is shown in figure 4.17. Based on this graph we can determine that adding the ability to limit outgoing infection attempts to ten dramatically reduces the total number of infected machines throughout the virus' lifespan if the percentage on infectable machines is low. However, if the percentage is greater than about 30 percent, nearly all the infectable nodes will still eventually become infected. Upon seeing this result, we were curious to discover whether reducing the number of outgoing infection attempts even further would significantly reduce the number of infected machines at higher infectable percentage rates.

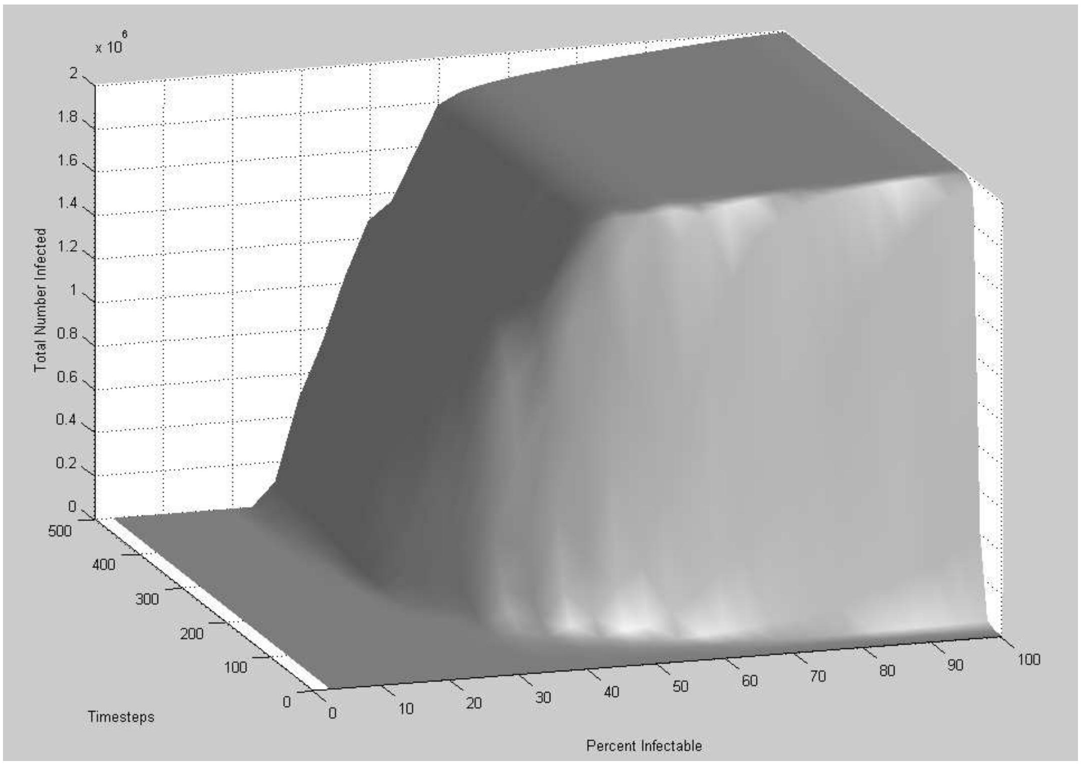


Figure 4.16: Graph of virus spread while only allowing 10 outgoing infection attempts per infected node

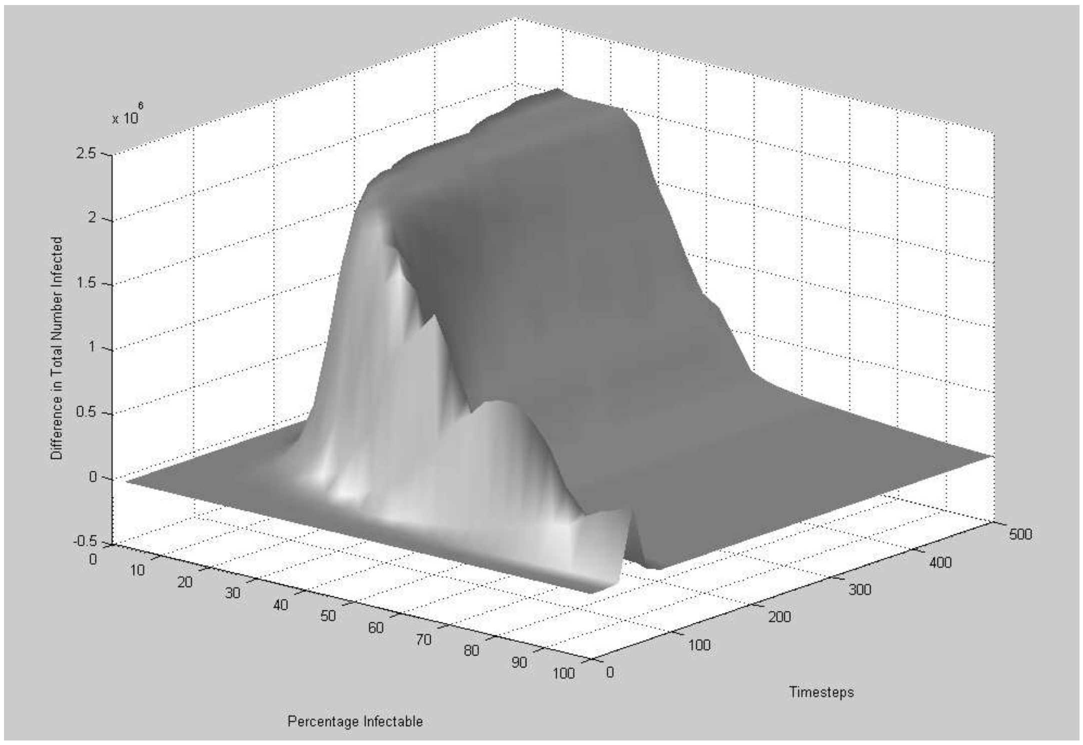


Figure 4.17: The difference in the total number of infected nodes between no suppression and limiting the outgoing infection attempts per infected node to 10

### 4.3.3 Five Outgoing Infections Per Node

Since limiting outgoing infection attempts to ten per infected node showed promise, we next tried limiting it to five attempts per infected node. The resulting graph is shown in figure 4.18. The difference between having no suppression and this graph are shown in figure 4.19. Just as we had hoped, changing the limit from ten to five infection attempts did reduce the number of infected machines at higher infectable percentage rates. In fact, in this scenario more than 50% of machines would need to be infectable for the virus to gain any ground at all in 500 or fewer timesteps.

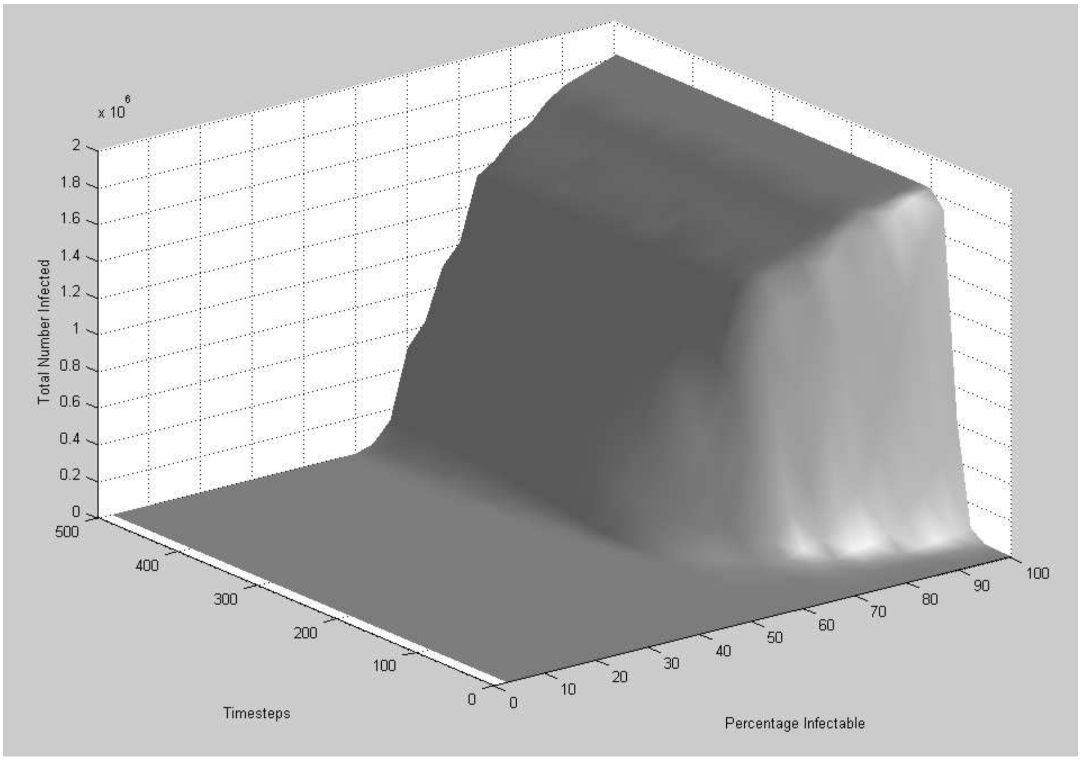


Figure 4.18: Graph of virus spread while only allowing 5 outgoing infection attempts per infected node

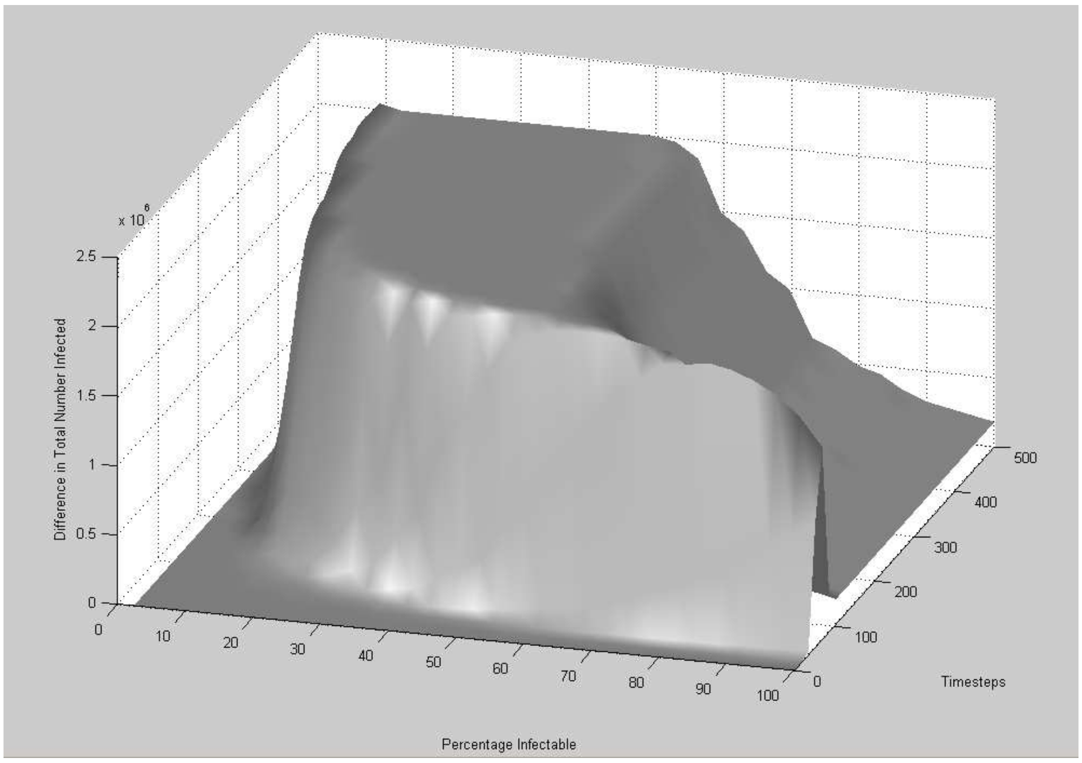


Figure 4.19: The difference in the total number of infected nodes between no suppression and limiting the outgoing infection attempts per infected node to 5

### 4.3.4 Three Outgoing Infections Per Node

To see if we could reduce the number of machines infected at even higher infectable percentages, we also simulated limiting the infection attempts to three per infected node. The results are shown in figure 4.20. Based on this graph, even at 100% of the nodes being infectable, fewer than 3,000 nodes are infected. It is important to keep in mind that the Z-axis on these graphs are not averaged, so fewer than 3,000 total nodes were infected over 20 runs of the experiment. The difference between not having a limit on the infection attempts and limiting them to three are shown in figure 4.21. Based on the graph in figure 4.20 it was no surprise to see that the difference very closely matched the graph with no suppression at all (figure 4.15).

Using Hephaestus we have shown that lossy detection can be an incredibly powerful way to slow the spread of a virus even without having signature-based anti-virus software installed on the potentially infected machines. It is important, but somewhat counter-intuitive, to note that even though some infection attempts do occur when a node first becomes infected the spread of the virus is greatly reduced due to the virus attempting to infect already infected nodes or uninfectable nodes. Therefore it is not necessary to block all virus spread attempts in order to slow the virus spread down enough to update anti-virus signatures and distribute them to all potentially infected machines.

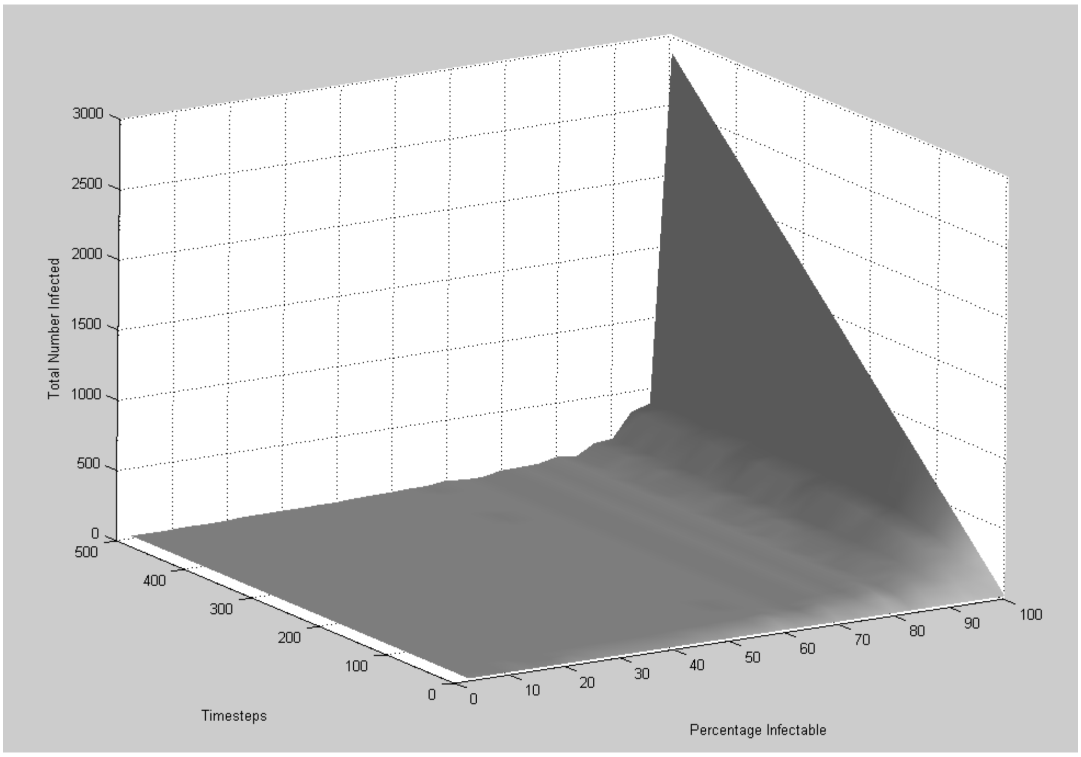


Figure 4.20: Graph of virus spread while only allowing 3 outgoing infection attempts per infected node

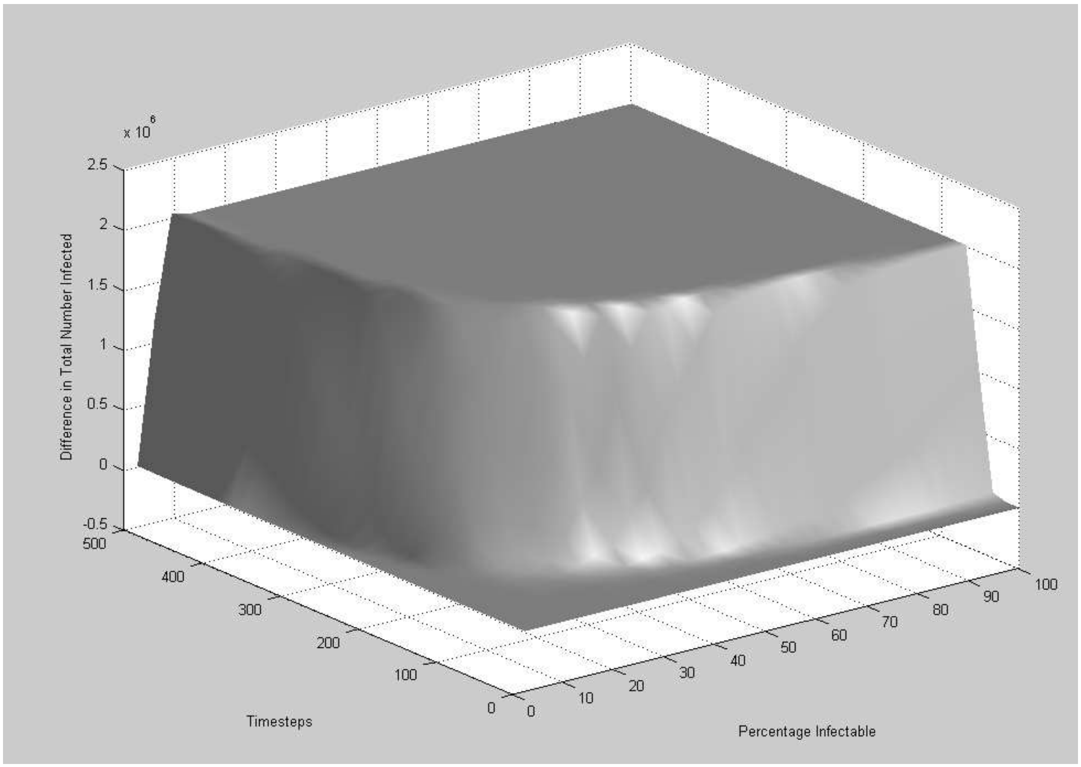


Figure 4.21: The difference in the total number of infected nodes between no suppression and limiting the outgoing infection attempts per infected node to 3

## 4.4 Tarpits

Tom Liston has created a tool called LaBrea to slow down or stop the spread of active worms[27]. It works by taking over unused IP addresses and creates “virtual machines” that respond to requests to that IP address. When it responds to the messages sent to the virtual address, it responds in such a way that the machine at the other end of the request gets stuck. LaBrea then can hold that connection open for a very long time so the machine at the other end is “stuck” for a long time[16].

It is fairly simple to simulate this tool using Hephaestus to observe the result of tarpits on various network configurations. The first thing we needed to do was create a new type of actor called a TarpitNode. This node type allows all incoming connections to succeed, but then never respond to infection attempts. This will cause the node attempting to infect the TarpitNode to have its connection stuck waiting for a response that will never occur.

We ran this experiment with four different configurations - 2% infectable, 5% infectable, 10% infectable, and 25% infectable. In all four configurations there were a total of 200,000 nodes. In each configuration we varied the percentage of tarpits in the network from 2.5% to 25% in 2.5% increments for a total of 10 configuration files in each of the four configurations. Each of these percentages will be identified by the variable $P$ in the equations below. The configuration file had the format shown below:

```
.\actors\InfectableNodeLibrary.dll,NI,1,30,50
.\actors\UnInfectableNodeLibrary.dll,NU,1,50,10
.\actors\TarpitNodeLibrary.dll,NT,500,500,500
```

where $NI$ was the number of infectable nodes (4,000 for the 2% configuration, 10,000 for the 5% configuration, etc), $NT$ was defined by the equation

$$NT = 200000 * (P/100) \tag{4.5}$$

and $NU$ was defined by the equation

$$NU = 200000 - NT - NI \tag{4.6}$$

The graphs of these experiments are shown in figures 4.22, 4.23, 4.24, and 4.25. In each case if more than 5 or 10 percent of the nodes in the network were tarpits, the spread of the virus was almost negligible until approximately 25% of the nodes being infectable. Even in that case, the spread of the virus was signficantly less than it would have been if there were no tarpits on the network.

In the graph of the 2% configuration (figure 4.22) there was an obvious hill near 17.5% tarpits that increases when we expected it to steadily decrease as the percentage of tarpits increased. There are other "jags" in the graph of the 2% configuration whereas the other graphs are much smoother. The reason for this is because of a combination of two factors. First, in the 2% graph the scale on the Z-axis is small so any variations in the number infected will be very obvious. Secondly, even though each experiment was run 20 times to help average out the variations in randomness in the simulations due to the actual random numbers chosen, if tarpits are selected for infection attempt early in the execution when there are few infected nodes the spread of the virus will not gain much ground at all. In other words, in this case 20 executions of the experiment was not enough to make the graph as smooth as we expected.

Running this experiment under Hephaestus verifies that using tarpits to slow or stop the spread of a malicious applications can be a very effective technique, although a relatively large number of virtual machines are required to fully stop the virus.

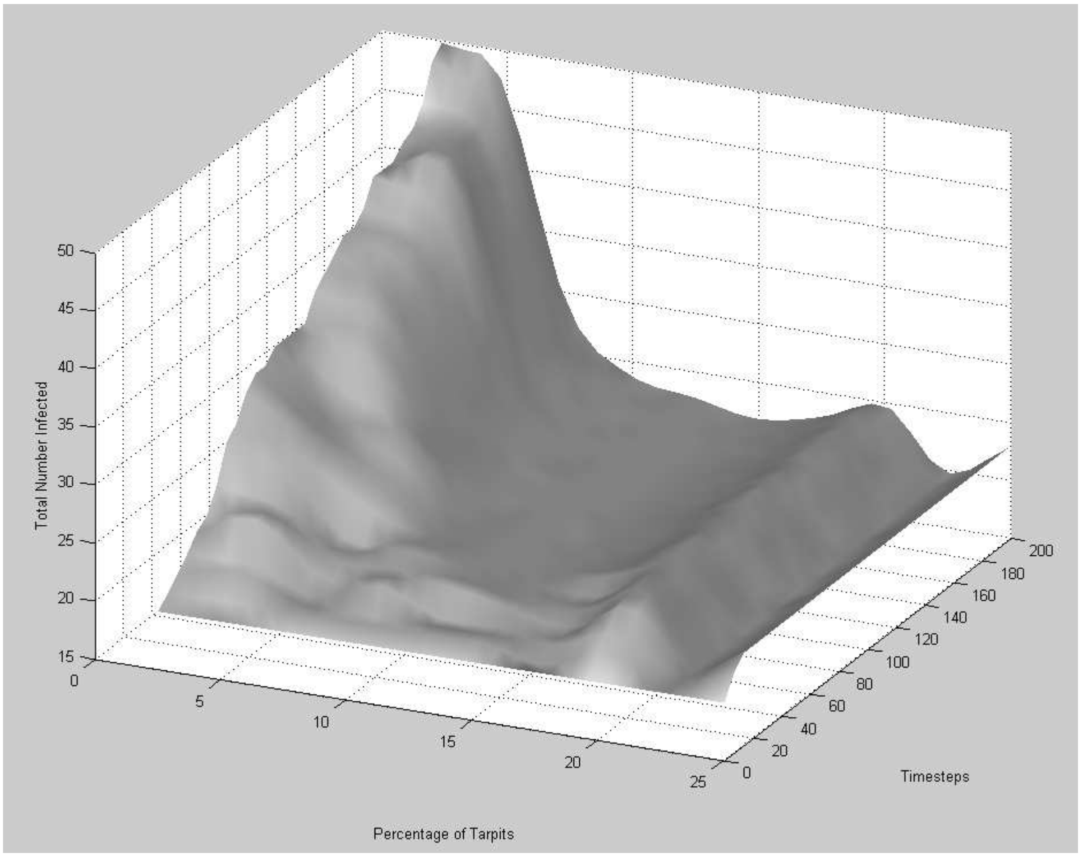


Figure 4.22: Graph of virus spread with tarpits and 2 percent of the nodes infectable

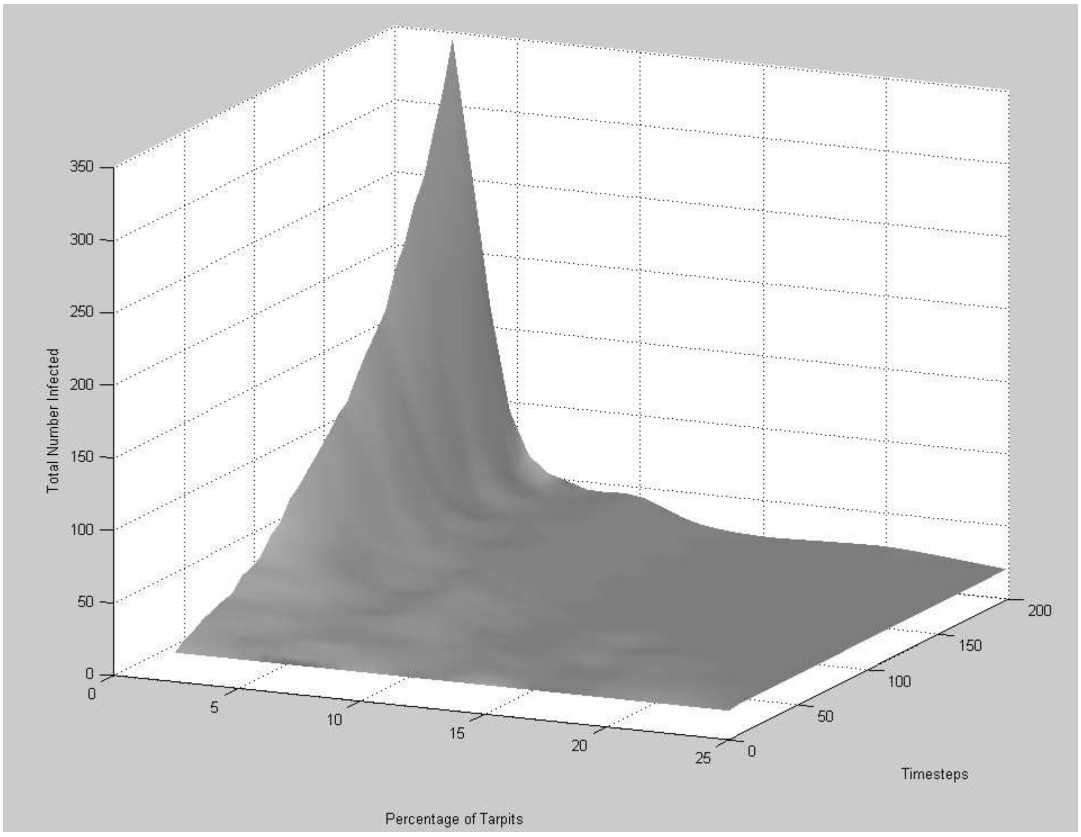


Figure 4.23: Graph of virus spread with tarpits and 5 percent of the nodes infectable

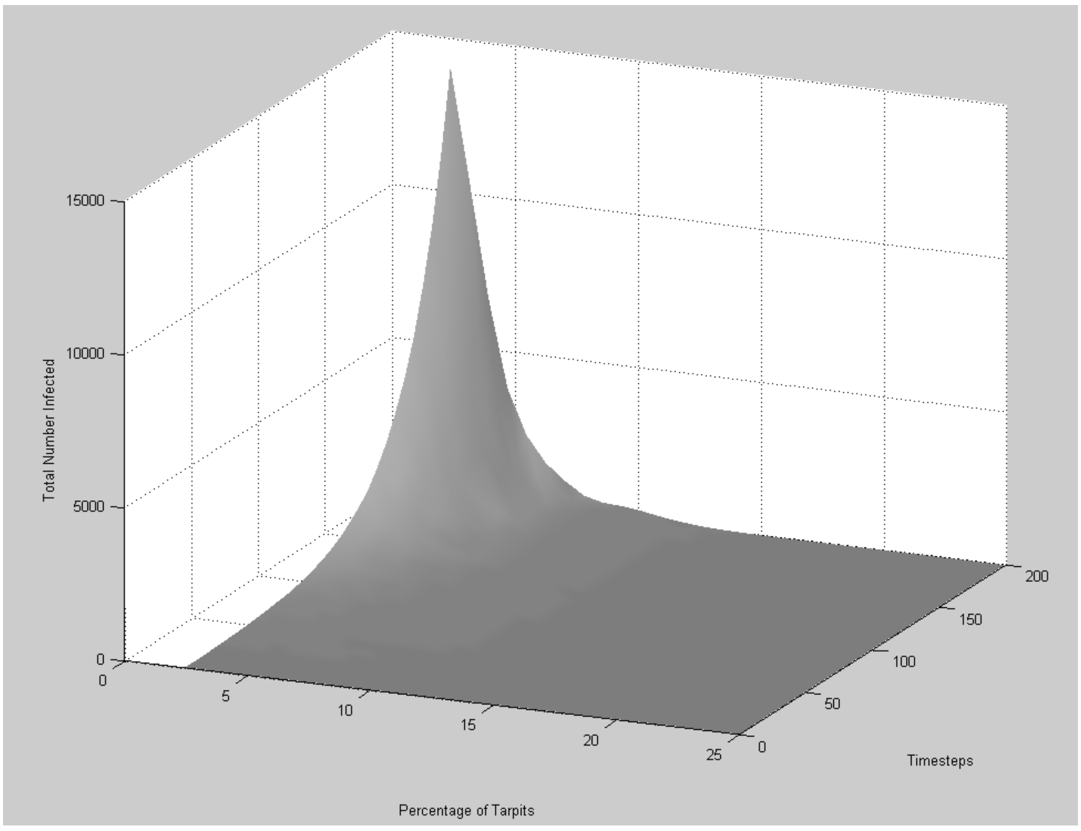


Figure 4.24: Graph of virus spread with tarpits and 10 percent of the nodes infectable

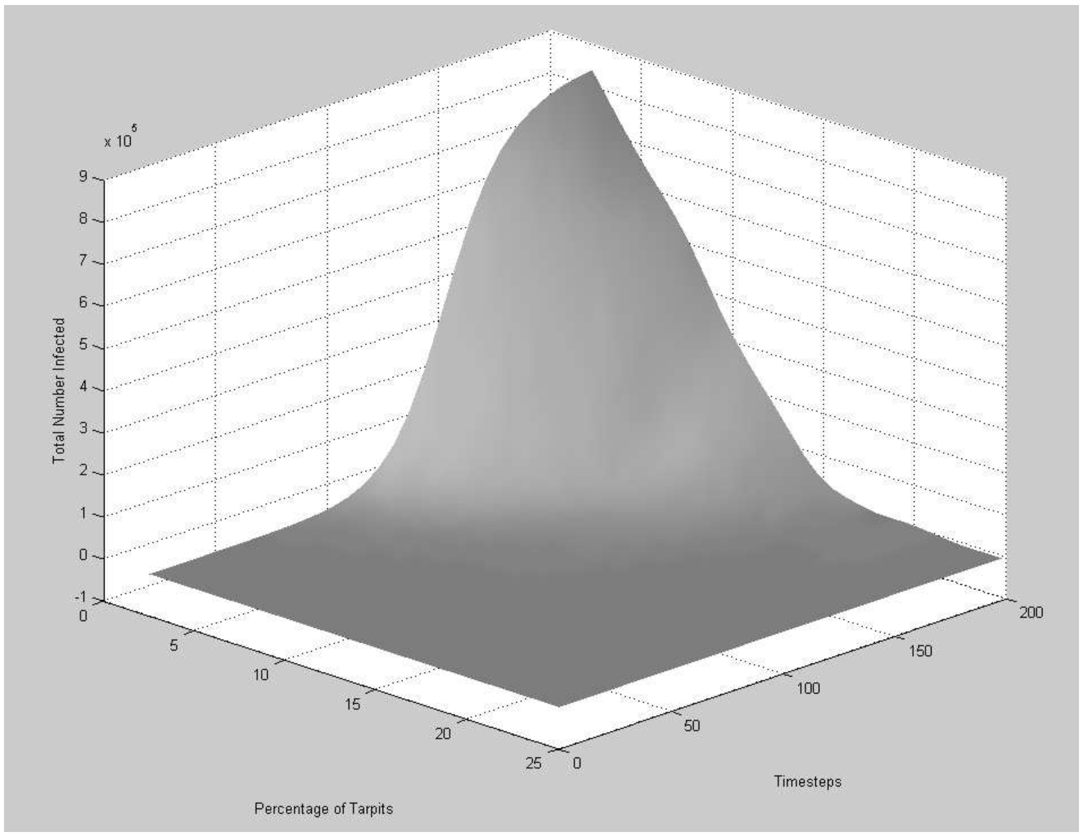


Figure 4.25: Graph of virus spread with tarpits and 25 percent of the nodes infectable

# Chapter 5

# Future Development Plans

Despite the power and flexibility of the Hephaestus simulation framework described in this thesis, it is not yet the complete simulator it can become. This section tries to outline tasks that should be taken into consideration when future contributors enhance the framework. Even as this is being written, a few of these tasks are already being implemented by Attila Ondi and Manan Pancholi at Florida Institute of Technology, and we expect that work to continue well into the future.

First, and most importantly, the current version of Hephaestus does not understand network topologies. It also assumes that any node can complete a connection or infection attempt to any other node in exactly one time step. Of course, in a real network this is rarely the case. The current version also assumes that each connection between two nodes has an unlimited bandwidth, so this needs to be addressed as well. Alleviating these shortcomings is the task Ondi and Pancholi are currently undertaking.

As part of adding capabilities to understand network topologies, Hephaestus also needs to have the ability to simulate network-based prevention techniques. The current version of Hephaestus is extremely capable of simulating host-based prevention techniques, but network-based techniques are still unimplemented. As a proof of concept experiment, "sensor" nodes can be placed in various locations throughout the network and attempt to slow or even stop the spread of a virus from one side of the sensor node to the other. This could in effect limit the spread of a virus to one large chunk of the network, rather than having it spread to all corners of the network.

The current version of Hephaestus does not have any concept of a death rate of infection or of a patch rate of infectable machines. Adding these two features could potentially add more realism to the numbers and graphs generated during simulations.

The concept of a "hitlist" does exist in the current version by the existence of an "initial infected count" parameter to the engine. Unfortunately the location of the machines in the hitlist is generated randomly. This is fine for the current version, but once network topology is accessible, this should be expanded to infect specific machines at different points in the network, as this could enhance the rate of spread of a virus.

Once Hephaestus is topology-aware, targeted scanning should be implemented. Currently all connections and infections are attempted towards randomly selected nodes. The simulated virus should be able to target specific nodes in the network. This would allow somewhat more realistic simulation of distributed denial-of-service attacks than what is currently available.

The random number generator chosen for random scanning is a very important piece of any simulator. As a teaching tool, if the random number generator was abstracted into a DLL, it would allow different random number generators that are potentially less reliable than the Mersenne Twister to be implemented to see how the random number generator used can affect the spread of a virus. For instance, we could observe changes in the results of the simulator if we use random scanning using the Mersenne Twister algorithm as compared to the rand() function in the Windows C-Runtime Library, or against a poorly seeded random number generator like the one that was used in the original Code Red.

Hephaestus is a good simulation framework in its current state, with the capability to expand and enhance the simulations it can perform. It is a good teaching tool, as well as a good way to simulate the applicability of many host-based prevention techniques. By adding just a few more features listed in this section, Hephaestus can become a great tool with uses in academia and industry.

# Chapter 6

# Summary

Malicious applications have become an increasingly frustrating and expensive problem for computer users, both at home and in corporate environments. Signature-based anti-virus software has become commonplace in an attempt to protect these users from attacks. It is not working, so something more needs to be done. If the solution to the anti-virus problem can not be solely signature based, we need to know more about the lifespan of a virus. We need to know how it attempts to attack other machines, how often it tries to attack other machines, and which machines will let the attack succeed.

Other researchers have gone before us and have given us a good starting point to understanding malicious code, but what is needed is a way to simulate prevention techniques other than by signature-based anti-virus software. This solution needs to be powerful and flexible, but still easy to use. That is exactly what Hephaestus was designed to accomplish.

## 6.1 Our Results And Contributions

We have developed a framework for performing simulations of malicious application spread over very large networks. In doing so, we have provided the capability to add new actor types to the simulation. This allows prevention measures to be simulated and instantly "plugged in" to the simulator framework. We have also allowed customization of the individual experiments through the use of configuration files. We have provided an easy-to-use Windows user interface with built-in graphing capability to control the framework. As a result we have developed a great teaching tool that professors can use to teach their students how to think about malicious application spread and how to better prevent that spread. We have also provided a command line interface to use to run multiple experiments unattended using batch files. We have also documented how to use the framework to run different experiments.

We used Hephaestus to verify that it produces results consistent with the Weaver simulator[19] when parameters consistent with that experiment were applied. We also verified the results of the Monoculture experiment performed by Dr. Richard Ford[26]. We then presented results of two powerful prevention techniques known as Lossy Detection and Tarpits and found them both to be very effective, enough so to easily slow the spread of a virus to the point where anti-virus signatures could be distributed before wide spread infection occurs.

## 6.2 Final Thoughts

The Hephaestus simulation framework is a fast, powerful, flexible framework for examining the effects of various prevention measures on a virus. Although much work has already been done on Hephaestus, there is much more development to be done before it can be called complete (see Chapter 5). Already some of that development has begun. It is our ultimate goal to have provided a simulator that will enable companies in industry to develop and test novel ideas on how to fix the malicious mobile code problem and reduce the $50+ billion in damage caused every year. With the right exposure to those companies, we believe we have accomplished this goal with Hephaestus.

# Appendix A

# Analysis of the Mersenne Twister Pseudorandom Number Generation Algorithm

In any malicious code spread simulator that uses random scanning to choose the next node to try to infect, it is crucial to have a good random number generator. If the random number generator used in the simulator does not generate truly random numbers we will not be able to accurately simulate the spread of that piece of malicious code.

But how does one ensure that numbers generated by a random number generator algorithm are good enough? According to Banks, Carson, and Nelson there are five tests to determine whether or not a random number generator generates truly random numbers - the Frequency, Runs, Autocorrelation, Gap, and Poker tests[28]. The random number generator algorithm used in Hephaestus, the Mersenne Twister Algorithm, was run through this series of tests as well as one extra test, the Completeness Test, before being allowed to be included in Hephaestus. The results of each of these tests for the algorithm follow after the explanation of the algorithm itself.

## A.1 The Mersenne Twister Pseudorandom Number Generation Algorithm

The Mersenne Twister algorithm was invented by Matsumoto and Nishimura and has several rather impressive properties as a random number generator. It has a period of $2^{19937} - 1$, which means that the sequence of numbers generated will only repeat after $2^{19937} - 1$ numbers has been generated. That number is well beyond the limits of the period needed by Hephaestus or almost any other application conceivable. The algorithm is also 623-dimensional equidistributed, which means that there is virtually no correlation between a generated number and its successor[29]. It is called the Mersenne Twister algorithm because of its period. The 19937 exponent in the period of $2^{19937} - 1$ is a Mersenne Prime (a number $n$ in which $2^n - 1$ is a prime number)[30]. This particular Mersenne Prime was discovered in 1971 by Tuckerman[30].

The algorithm is basically a linear-feedback shift register (a shift register whose input is the exclusive-or of some of its outputs[31]) with a seed value of 19937 bits stored in an array of 624 32-bit integer fields. This leaves 31 bits unused, and these bits are migrated throughout the array as more numbers are generated so to produce a higher period in the generated numbers[32]. Source code for the implementation is below

```
/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0df   /* constant vector a */
#define UPPER_MASK 0x80000000 /* most significant w-r bits */
#define LOWER_MASK 0x7fffffff /* least significant r bits */

/* Tempering parameters */
#define TEMPERING_MASK_B 0x9d2c5680
#define TEMPERING_MASK_C 0xefc60000
#define TEMPERING_SHIFT_U(y)  (y >> 11)
#define TEMPERING_SHIFT_S(y)  (y << 7)
#define TEMPERING_SHIFT_T(y)  (y << 15)
#define TEMPERING_SHIFT_L(y)  (y >> 18)

#define DIVISOR 0xFFFFFFFF

static unsigned long mt[N]; /* the array for the state vector  */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */
static unsigned long range; /* the range value for the upper bound of random numbers */
static unsigned long shiftright; /* Convenience - only need to calculate when we
                                    init range */

/* initializing the array with a NONZERO seed */
void sgenrand(unsigned long seed)
{
    /* setting initial seeds to mt[N] using         */
    /* the generator Line 25 of Table 1 in          */
    /* [KNUTH 1981, The Art of Computer Programming */
    /*    Vol. 2 (2nd Ed.), pp102]                  */
    mt[0]= seed & 0xffffffff;
    for (mti=1; mti<N; mti++)
        mt[mti] = (69069 * mt[mti-1]) & 0xffffffff;
}

/****************************************************
 * This function generates a random number between 0
 * and 2^32 - 1
 ****************************************************/
unsigned long genrand()
{
    unsigned long y;
    static unsigned long mag01[2]={0x0, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */
    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if sgenrand() has not been called, */
            sgenrand(4357); /* a default initial seed is used   */

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1];

        mti = 0;
    }

    y = mt[mti++];
    y ^= TEMPERING_SHIFT_U(y);
    y ^= TEMPERING_SHIFT_S(y) & TEMPERING_MASK_B;
    y ^= TEMPERING_SHIFT_T(y) & TEMPERING_MASK_C;
    y ^= TEMPERING_SHIFT_L(y);

    return y;
}
```

## A.2 Frequency Test

The Frequency Test tests the uniformity of the generated numbers using the Kolmogorov-Smirnov test[33]. This test measures how well the distribution of the generated number matches the theoretical distribution of true randomness. The results are then compared to a critical value which will either allow us to reject the randomness of the algorithm or continue to try the other tests on more random numbers. Note that the results can only allow us to reject the algorithm. Not rejecting the randomness of the algorithm does not imply that we can blindly reason that the algorithm produces truly random numbers. This test, as well as the other four tests, only allow us to reject some obviously non-random algorithms. There is no way to guarantee that numbers are truly random; in fact, that the numbers are computed by a defined algorithm guarantees the generated numbers are not truly random. The best an algorithm can hope to accomplish is to simulate randomness well enough that implementers of the algorithm are satisfied.

There are six steps to performing the Frequency Test, and they are listed below:

1. Generate some random numbers

2. Sort the generated numbers from lowest to highest.
3. Compute D$^+$ and D$^-$ using the equations:

$$D^+ = \max_{1 \leq i \leq N} \left\{ \frac{i}{N} - R_{(i)} \right\} \tag{A.1}$$

$$D^- = \max_{1 \leq i \leq N} \left\{ R_{(i)} - \frac{i-1}{N} \right\} \tag{A.2}$$

4. Compute D as follows:

$$D = max(D^+, D^-) \tag{A.3}$$

5. Determine the critical value D$_\alpha$ for the specified significance level $\alpha$ and sample size $N$
6. If D is greater than D$_\alpha$ we reject the randomness of the algorithm, otherwise we continue to another test.

The numbers generated for the Frequency Test are listed in table A.1. The same numbers, when sorted from least to greatest are listed in table A.2. The value of D$^+$ for these numbers is 0.120183978. The value of D$^-$ for these numbers is 0.153916219. D then is equal to D$^-$. Given a sample size of 35 and an $\alpha$ value of 0.05, D$_\alpha$ is 0.23[34]. Since D is less than D$_\alpha$, the Mersenne Twister's randomness can not be rejected based on the Frequency Test.

| | | | | |
|---|---|---|---|---|
| 0.328135132 | 0.094007755 | 0.777706933 | 0.788982696 | 0.582487648 |
| 0.601514765 | 0.096415540 | 0.735720048 | 0.822673164 | 0.672452017 |
| 0.616672127 | 0.079329750 | 0.214444376 | 0.692034661 | 0.805573514 |
| 0.060424835 | 0.507170467 | 0.604370462 | 0.100203250 | 0.248727390 |
| 0.632502295 | 0.167439384 | 0.733779989 | 0.900476184 | 0.729695989 |
| 0.657998779 | 0.123972001 | 0.857694353 | 0.284099798 | 0.284572988 |
| 0.739053965 | 0.068633097 | 0.510586272 | 0.783697488 | 0.710714413 |

Table A.1: Random Numbers Generated for the Frequency Test

| | | | | |
|---|---|---|---|---|
| 0.060424835 | 0.068633097 | 0.079329750 | 0.094007755 | 0.096415540 |
| 0.100203250 | 0.123972001 | 0.167439384 | 0.214444376 | 0.248727390 |
| 0.284099798 | 0.284572988 | 0.328135132 | 0.507170467 | 0.510586272 |
| 0.582487648 | 0.601514765 | 0.604370462 | 0.616672127 | 0.632502295 |
| 0.657998779 | 0.672452017 | 0.692034661 | 0.710714413 | 0.729695989 |
| 0.733779989 | 0.735720048 | 0.739053965 | 0.777706933 | 0.783697488 |

|  |  |  |  |  |
|---|---|---|---|---|
| 0.788982696 | 0.805573514 | 0.822673164 | 0.857694353 | 0.900476184 |

Table A.2: Sorted Numbers for the Frequency Test

## A.3 Runs Test

There are many sets of generated random numbers that may pass the Frequency Test but are not uniformly distributed. For instance, truly random numbers will be unlikely to have a large number of strictly increasing or strictly decreasing values generated consecutively. Other similar potential problems exist if a large sequence of numbers is either above or below the mean value, and also if the length of each set of strictly increasing or decreasing numbers is longer than would naturally appear. All these occurrences are unlikely to appear with truly random numbers.

The Runs Test is really a set of three tests that test for each of these conditions in a set of generated random numbers. Those three tests are demonstrated for the Mersenne Twister algorithm in the next few sections. In each of these tests we will compare the value we calculate from the generated numbers to a theoretical value to see if we can reject the randomness of the Mersenne Twister algorithm. Also in each of these tests the generated numbers used for the tests are the same numbers as shown in Table A.1.

### A.3.1 Number of Runs Up and Runs Down

A run up is a sequence of numbers in the number set that increase from one to the next number. A run down is a sequence of numbers in the set that decrease from one to the next number. The number of increases or decreases in each run is called the run length. The first of the three Runs Tests simply count the number of runs and compares that number to a theoretical expected value for the number of runs.

If $N$ is the number of generated numbers, there can only be at most $N - 1$ runs in the set of numbers. The expected mean value of the sequence $\mu_a$ for a set of numbers with $a$ runs is given by the equation:

$$\mu_a = \frac{2N - 1}{3} \tag{A.4}$$

and the variance $\sigma_a^2$ is given by the equation:

$$\sigma_a^2 = \frac{16N - 29}{90} \tag{A.5}$$

For large values of $N$ ($> 20$) the distribution of $a$ can be approximated as a normal distribution, which means that a standardized normal test statistic $Z_0$ can be defined as

$$Z_0 = \frac{a - \mu_a}{\sigma_a} \tag{A.6}$$

which when expanded becomes

$$Z_0 = \frac{a - [\frac{2N-1}{3}]}{\sqrt{\frac{16N-29}{90}}} \tag{A.7}$$

If $Z_0$ is between -1.96 and 1.96 ($Z_{\alpha/2}$ for $\alpha = 0.05$)[28], then we can not reject the algorithm based on this test. For our generated numbers there were 10 runs up and 11 runs down. The value of $Z_0$ is -0.82338697, so we can not reject the algorithm based on this run of the test.

### A.3.2 Number of Runs Above and Below the Mean

This test compares the number of runs above and below the mean to a theoretical value to see if the "random" numbers generated are sufficiently random. If $n_1$ is the number of values above the mean value of the set and $n_2$ is the number of values below the mean, then there is between 1 and $N$ possible runs in the data set where $N = n_1 + n_2$. We will call the total number of runs $b$ The new equations for the mean $\mu_b$, variance $\sigma_b^2$, and test statistic $Z_0$ become

$$\mu_b = \frac{2n_1n_2}{N} + \frac{1}{2} \tag{A.8}$$

$$\sigma_b^2 = \frac{2n_1n_2(2n_1n_2 - N)}{N^2(N-1)} \tag{A.9}$$

$$Z_0 = \frac{b - \frac{2n_1n_2}{N} - \frac{1}{2}}{\sqrt{\frac{2n_1n_2(2n_1n_2-N)}{N^2(N-1)}}} \tag{A.10}$$

For the generated numbers we get a mean value of 0.499999995 and a $Z_0$ value of 0.426097832. This value is within the critical theoretical limits of -1.96 to 1.96, so we can not reject the algorithm based on the results of this test.

### A.3.3 Length of Runs

This test compares the run length of each run against the expected value for truly random numbers. If we call $Y_i$ the number of runs of length $i$ in a sequence of $N$ numbers, the expected value of $Y_i$ becomes

$$E(Y_i) = \frac{2}{(i+3)!}[N(i^2 + 3i + 1) - (i^3 + 3i^2 - i - 4)] \tag{A.11}$$

when $i \leq N$ - 2, and

$$E(Y_i) = \frac{2}{N!} \tag{A.12}$$

when $i = N$ - 1 for the number of runs up and down.

These values are compared against the theoretical values using a chi-square test[35], and the test statistic $\chi_0^2$ is given by the equation

$$\chi_0^2 = \sum_{i=1}^{N-1} \frac{[O_i - E(Y_i)]^2}{E(Y_i)} \tag{A.13}$$

where $O_i$ is the observed number of runs of length $i$ in the number set.

For the numbers generated in A.1, $\chi_0^2$ equates to 2.41724675, and the critical value of $\chi_0^2$ is 3.84, so we can not reject the randomness of the algorithm for this test. Since we can not reject the randomness of the algorithm using any of the three Runs Tests we can not reject the randomness of the algorithm using the Runs Test.

## A.4 Autocorrelation Test

It is possible that numbers generated can seem random while having a correlation between some of the generated numbers. For instance if the third, sixth, ninth, twelfth, etc. numbers in the set are all very large numbers, or are very small numbers, or some have some other suspicious relationship then the numbers generated might not be sufficiently random for many applications. The Autocorrelation Test checks for relationships like these. Unfortunately there are hundreds of tests to run to ensure there are no such relationships even for relatively small number sets, so we will show just one in this section.

In this example we will demonstrate there is no autocorrelation between every five numbers in a generated set of numbers starting with the third number in the set. Some variables and equations need to be defined to do run this test. We will call $m$ the lag between each number we want to test. In our case this is 5. The starting number we will call $i$, which is 3. The autocorrelation we are interested in is between the numbers $R_i$, $R_{i+m}$, $\ldots$, $R_{i+(M+1)m}$, where $R_j$ is the $j^{th}$ number in the set of generated numbers. $M$ is the largest integer such that $i + (M + 1)m \leq N$. $N$ is the total number of values in the generated number set.

Our test statistic can be defined by the equation

$$Z_0 = \frac{\hat{\rho}_{im}}{\sigma_{\hat{\rho}_{im}}} \tag{A.14}$$

where $\hat{\rho}_{im}$ and $\sigma_{\hat{\rho}_{im}}$ are defined as

$$\hat{\rho}_{im} = \frac{1}{M+1}[\sum_{k=0}^{M} R_{i+km}R_{i+(k+1)m}] - 0.25 \tag{A.15}$$

and

$$\sigma_{\hat{\rho}_{im}} = \frac{\sqrt{13M+7}}{12(M+1)} \tag{A.16}$$

Table A.3 contains the numbers generated for the Autocorrelation Test. Since we are using values of 5 for $m$ and 3 for $i$, $\hat{\rho}_{35} = 0.0096$ and $\sigma_{\hat{\rho}_{im}} = 0.0681$. $Z_0$ therefore equates to 0.1410. The critical value for $Z_0$ with $\alpha = 0.05$ is 1.96. $Z_0$ is less than the critical value, so we can not reject the Mersenne Twister algorithm based on this small part of a complete Autocorrelation Test.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0.42 | 0.12 | 0.99 | 0.74 | 0.76 | 0.12 | 0.94 | 0.86 | 0.78 | 0.10 |
| 0.27 | 0.88 | 0.07 | 0.64 | 0.77 | 0.12 | 0.31 | 0.80 | 0.21 | 0.93 |
| 0.93 | 0.84 | 0.15 | 0.36 | 0.36 | 0.94 | 0.08 | 0.65 | 0.90 | 0.48 |
| 0.24 | 0.32 | 0.03 | 0.50 | 0.88 | 0.63 | 0.92 | 0.73 | 0.98 | 0.98 |
| 0.76 | 0.54 | 0.00 | 0.58 | 0.98 | 0.75 | 0.52 | 0.50 | 0.79 | 0.08 |
| 0.57 | 0.88 | 0.37 | 0.28 | 0.51 | 0.07 | 0.50 | 0.34 | 0.79 | 0.38 |
| 0.07 | 0.00 | 0.91 | 0.70 | 0.21 | 0.99 | 0.33 | 0.61 | 0.16 | 0.36 |
| 0.42 | 0.51 | 0.06 | 0.01 | 0.08 | 0.23 | 0.61 | 0.49 | 0.71 | 0.65 |
| 0.95 | 0.76 | 0.06 | 0.33 | 0.83 | 0.51 | 0.26 | 0.46 | 0.01 | 0.10 |
| 0.48 | 0.59 | 0.23 | 0.17 | 0.97 | 0.09 | 0.31 | 0.35 | 0.48 | 0.48 |

Table A.3: Numbers Generated for the Autocorrelation Test

## A.5 Gap Test

The Gap Test is used to check the significance or the interval between instances of a particular digit in a set of generated numbers. For instance, given a set of numbers: 4, 1, 3, 5, 1, 7, 2, 8, 2, 0, 7, 9, 1, 3 the gap length between instances of the digit 3 is 10. In general, the probability of a gap of a given length is given by the equation

$$P(\text{t followed by exactly x non-t digits}) = (0.9)^x(0.1) \tag{A.17}$$

and the probability that a gap is less than or equal to $x$ for truly random numbers is given by

$$F(x) = 0.1\sum_{n=0}^{x}(0.9)^n = 1 - 0.9^{x+1} \tag{A.18}$$

There numbers generated for the Gap Test are listed in Table A.4. The number of gaps for each digit 0 through 9 are listed in Table A.5. The Gap Test data for the Mersenne Twister algorithm is listed in Table A.6. The number of gaps in the data is the sample size minus the number of digits, or 110-10=100. Using an $\alpha$ value of 0.05, the critical theoretical value $D_\alpha$ is 0.136[34]. $D$ for the generated numbers is given by $\max |F(x) - S_N(x)|$, which equates to 0.0653020189. Therefore we can not reject the Mersenne Twister algorithm based on the Gap Test.

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 1 | 9 | 9 | 1 | 9 | 1 | 3 | 0 | 8 | 9 | 1 | 3 | 2 | 1 | 4 | 4 | 1 | 8 | 6 | 3 | 4 |
| 0 | 6 | 7 | 9 | 9 | 6 | 0 | 7 | 9 | 6 | 6 | 9 | 1 | 7 | 4 | 3 | 6 | 0 | 6 | 4 | 9 | 4 |
| 0 | 0 | 8 | 2 | 4 | 7 | 2 | 4 | 5 | 6 | 0 | 0 | 1 | 2 | 7 | 6 | 8 | 8 | 9 | 0 | 4 | 6 |
| 3 | 5 | 0 | 1 | 6 | 7 | 2 | 2 | 1 | 3 | 4 | 6 | 6 | 3 | 2 | 6 | 4 | 8 | 0 | 3 | 0 | 6 |
| 0 | 7 | 8 | 4 | 1 | 8 | 7 | 1 | 2 | 9 | 4 | 0 | 3 | 0 | 7 | 8 | 2 | 1 | 0 | 7 | 2 | 4 |

Table A.4: Numbers Generated for the Gap Test

| Digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Number of Gaps | 15 | 12 | 9 | 8 | 13 | 2 | 14 | 9 | 8 | 10 |

Table A.5: Numbers Generated for the Gap Test

| *GapLength* | *Frequency* | *Relative Frequency* | *CumulativeRelative Frequency* | $F(x)$ | $\lvert F(x) - S_N(x)\rvert$ |
|---|---|---|---|---|---|
| 0- 3 | 31 | 0.31 | 0.31 | 0.3439 | 0.0339 |
| 4- 7 | 31 | 0.31 | 0.62 | 0.5695 | 0.0505 |
| 8-11 | 12 | 0.12 | 0.74 | 0.7176 | 0.0224 |
| 12-15 | 14 | 0.14 | 0.88 | 0.8147 | 0.0653 |
| 16-19 | 5 | 0.05 | 0.93 | 0.8784 | 0.0516 |
| 20-23 | 2 | 0.02 | 0.95 | 0.9202 | 0.0298 |
| 24-27 | 1 | 0.01 | 0.96 | 0.9477 | 0.0123 |
| 28-31 | 1 | 0.01 | 0.97 | 0.9657 | 0.0043 |
| 32-35 | 2 | 0.02 | 0.99 | 0.9775 | 0.0125 |
| 36-39 | 0 | 0.00 | 0.99 | 0.9852 | 0.0048 |
| 40-43 | 0 | 0.00 | 0.99 | 0.9903 | 0.0003 |
| 44-47 | 0 | 0.00 | 0.99 | 0.9936 | 0.0036 |
| 48-51 | 1 | 0.01 | 1.00 | 0.9958 | 0.0042 |
| 52-55 | 0 | 0.00 | 1.00 | 0.9973 | 0.0027 |
| 56-59 | 0 | 0.00 | 1.00 | 0.9982 | 0.0018 |
| 60-63 | 0 | 0.00 | 1.00 | 0.9988 | 0.0012 |
| 64-67 | 0 | 0.00 | 1.00 | 0.9992 | 0.0008 |
| 68-71 | 0 | 0.00 | 1.00 | 0.9995 | 0.0005 |
| 72-75 | 0 | 0.00 | 1.00 | 0.9997 | 0.0003 |
| 76-79 | 0 | 0.00 | 1.00 | 0.9998 | 0.0002 |
| 80-83 | 0 | 0.00 | 1.00 | 0.9999 | 0.0001 |
| 84-87 | 0 | 0.00 | 1.00 | 0.9999 | 0.0001 |
| 88-91 | 0 | 0.00 | 1.00 | 0.9999 | 0.0001 |
| 92-95 | 0 | 0.00 | 1.00 | 1.0000 | 0.0000 |
| 96-99 | 0 | 0.00 | 1.00 | 1.0000 | 0.0000 |

Table A.6: Gap Test Data

## A.6 Poker Test

The Poker Test is designed to detect large amount of digit repetition in a single number. For instance the following numbers look suspicious as they all have a pair of like digits: 0.255, 0.577, 0.331, 0.414, 0.828, 0.909, 0.303, 0.001. This might be an indication that numbers are not as random as we would like them to be.

Given a three-digit number, there are only three possibilities for digit combinations, and they are listed below.

1. The individual digits can all be different.
2. The individual digits can all be the same.
3. There can be one pair of like digits.

The probability of each of these cases is given by the following:

$$P(3 \text{ different digits}) = P(2^{nd}\neq 1^{st})\times P(3^{rd}\neq 1^{st}\text{and}2^{nd}) = (0.9)(0.8) = 0.72 \quad \text{(A.19)}$$

$$P(3 \text{ like digits}) = P(2^{nd} = 1^{st})\times P(3^{rd} = 1^{st}) = (0.1)(0.1) = 0.01 \quad \text{(A.20)}$$

$$P(\text{exactly 1 pair}) = 1 - P(\text{three different digits}) - P(\text{three like digits}) = 0.27 \quad \text{(A.21)}$$

The numbers used in the Poker Test are listed in Table A.7. The test is a chi-square[35] similar to the Number of Runs Up and Down Test shown earlier. Using a $\alpha$ value of 0.05, $\chi^2_{0.05,2} = 5.99$. The calculations for our data are shown in Table A.8. $\chi^2$ is 3.70, which is less than 5.99 so we can not reject the algorithm based on the Poker Test.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 336 | 96 | 796 | 807 | 596 | 615 | 98 | 753 | 842 | 688 |
| 631 | 81 | 219 | 708 | 824 | 61 | 519 | 618 | 102 | 254 |
| 647 | 171 | 751 | 922 | 747 | 673 | 126 | 878 | 290 | 291 |
| 756 | 70 | 522 | 802 | 727 | 384 | 193 | 956 | 263 | 27 |
| 402 | 704 | 506 | 739 | 586 | 784 | 785 | 614 | 433 | 7 |
| 470 | 831 | 791 | 600 | 977 | 886 | 416 | 403 | 635 | 71 |
| 459 | 704 | 303 | 954 | 231 | 415 | 58 | 401 | 279 | 638 |
| 305 | 56 | 6 | 731 | 564 | 169 | 795 | 269 | 955 | 490 |
| 134 | 295 | 340 | 408 | 53 | 12 | 70 | 190 | 489 | 393 |
| 572 | 527 | 916 | 762 | 609 | 971 | 50 | 271 | 670 | 409 |

Table A.7: Numbers Generated for the Poker Test

| Combination | Observed (O$_i$) | Expected (E$_i$) | $\frac{(O_i-E_i)^2}{E_i}$ |
|---|---|---|---|
| Three different digits | 80 | 72 | 0.89 |
| Three like digits | 0 | 1 | 1.00 |
| Three different digits | 20 | 27 | 1.81 |
| | 100 | 100 | 3.70 |

Table A.8: Calculations for the Poker Test

## A.7 Completeness Test

A final test to complete to ensure that the numbers generated are sufficiently random for most applications is to ensure that each and every number in a specified range will be generated eventually by the algorithm. This test requires no mathematical formula to interpret the results of this test since it is so straightforward, so we simply set a range of 100000 numbers and generated hundreds of thousands of numbers ensuring that every number from 0 to 99999 is generated.

In the first run of this test every number in the range was generated within 1346457 total numbers generated. In the second run of this test every number in the range was generated within 1163179 total numbers generated. In the third and final run of this test every number in the range was generated within 1294346 total numbers generated. Since every number in the range was generated in all three runs of the Completeness Test, we can not reject the randomness of the Mersenne Twister based on the Completeness Test.

## A.8 Conclusions

We have shown the results of running the Mersenne Twister algorithm under 6 different tests to help ensure that the numbers generated by the algorithm will simulate truly random numbers well enough for inclusion in Hephaestus. The Frequency Test, Runs Test, Autocorrelation Test, Gap Test, Poker Test, and Completeness Test were unable to provide evidence that the algorithm had serious flaws in the randomness of the numbers generated by it. Since we can not reject the randomness of the numbers generated by the Mersenne Twister algorithm based on any of the tests listed in this appendix, we felt safe to use this algorithm in Hephaestus when it needs to get random numbers during the execution of the simulations.

## A.9 Source Code Used for the Randomness Tests

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <vector>

using namespace std;

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0df   /* constant vector a */
#define UPPER_MASK 0x80000000 /* most significant w-r bits */
#define LOWER_MASK 0x7fffffff /* least significant r bits */
```

```
/* Tempering parameters */
#define TEMPERING_MASK_B 0x9d2c5680
#define TEMPERING_MASK_C 0xefc60000
#define TEMPERING_SHIFT_U(y)  (y >> 11)
#define TEMPERING_SHIFT_S(y)  (y << 7)
#define TEMPERING_SHIFT_T(y)  (y << 15)
#define TEMPERING_SHIFT_L(y)  (y >> 18)

#define DIVISOR 0xFFFFFFFF

static unsigned long mt[N]; /* the array for the state vector  */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */
static unsigned long range; /* the range value for the upper bound of random numbers */
static unsigned long shiftright; /* Convenience - only need to calculate when we
                                    init range */

/* initializing the array with a NONZERO seed */
void sgenrand(unsigned long seed)
{
    /* setting initial seeds to mt[N] using         */
    /* the generator Line 25 of Table 1 in          */
    /* [KNUTH 1981, The Art of Computer Programming */
    /*    Vol. 2 (2nd Ed.), pp102]                  */
    mt[0]= seed & 0xffffffff;
    for (mti=1; mti<N; mti++)
        mt[mti] = (69069 * mt[mti-1]) & 0xffffffff;
}

/****************************************************
 * This function generates a random number between 0
 * and 2^32 - 1
 ****************************************************/
unsigned long genrand()
{
    unsigned long y;
    static unsigned long mag01[2]={0x0, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if sgenrand() has not been called, */
            sgenrand(4357); /* a default initial seed is used   */

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1];

        mti = 0;
    }

    y = mt[mti++];
    y ^= TEMPERING_SHIFT_U(y);
    y ^= TEMPERING_SHIFT_S(y) & TEMPERING_MASK_B;
    y ^= TEMPERING_SHIFT_T(y) & TEMPERING_MASK_C;
    y ^= TEMPERING_SHIFT_L(y);

    return y;
}

void initmyrand(unsigned long myrange)
{
    // Calculate what value of n works well...
    int i;
    myrange = myrange - 1;
    range = myrange;
    for (i=0; i<32; i++) {
        if (myrange < pow (2, i))
        {
            break;
        }
    }
    shiftright = 0xffffffff;
    shiftright = (32 - i);
}

unsigned long myrand()
{
    unsigned long liMyLong;
    liMyLong = 0xffffffff;
    do {
       liMyLong = genrand();
       liMyLong = liMyLong >> shiftright;
    } while (liMyLong > range);
    return (liMyLong);
}

void printNumbers(vector<double> &numArray)
{
    vector<double>::iterator iter = numArray.begin();

    for (int i = 0; i < 7; i++)
    {
        for (int j = 0; j < 5; j++)
        {
            if (iter != numArray.end())
            {
                if (j > 0)
                    printf("\t");
                printf ("%0.9f", *iter);
            }

            iter++;
        }
        printf("\n");
    }
}

void printThreePrecisionNumbers(vector<unsigned long> &numArray)
{
    vector<unsigned long>::iterator iter = numArray.begin();

    for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            if (iter != numArray.end())
            {
                if (j > 0)
                    printf("\t");
                printf ("%3d", *iter);
            }

            iter++;
        }
        printf("\n");
    }
}

void printTwoPrecisionNumbers(vector<double> &numArray)
{
    vector<double>::iterator iter = numArray.begin();

    for (int i = 0; i < 10; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            if (iter != numArray.end())
            {
                if (j > 0)
                    printf("\t");
                printf ("%0.2f", *iter);
            }

            iter++;
        }
        printf("\n");
    }
}

void printIntegers(vector<unsigned long> &intArray)
{
    vector<unsigned long>::iterator iter = intArray.begin();

    for (int i = 0; i < 5; i++)
    {
        for (int j = 0; j < 22; j++)
        {
            if (iter != intArray.end())
            {
                if (j > 0)
                    printf(" ");
                printf ("%u", *iter);
            }

            iter++;
        }
        printf("\n");
    }
}

void generateNumbers(vector<double> &numArray)
{
    sgenrand((long)time(NULL));
    for (int i = 0; i < 35; i++)
    {
        unsigned long random = genrand();
        double dbl = (double)random / (double)DIVISOR;
        numArray.push_back(dbl);
    }
}

void generateThreePrecisionNumbers(vector<unsigned long> &numArray)
{
    sgenrand((long)time(NULL));
    initmyrand(1000);
    for (int i = 0; i < 100; i++)
    {
        unsigned long random = myrand();
        numArray.push_back(random);
    }
}

void generateTwoPrecisionNumbers(vector<double> &numArray)
{
    sgenrand((long)time(NULL));
    initmyrand(100);
    for (int i = 0; i < 100; i++)
    {
        unsigned long random = myrand();
        double result = random / 100.0;
        numArray.push_back(result);
    }
}

void generateIntegers(vector<unsigned long> &intArray)
{
    sgenrand((long)time(NULL));
    initmyrand(10);
    for (int i = 0; i < 110; i++)
    {
        unsigned long random = myrand();
        intArray.push_back(random);
    }
}

void sortNumbers(vector<double> &unorderedNums, vector<double> &orderedNums)
{
    // use a simple selection sort
    orderedNums.clear();

    orderedNums.push_back(unorderedNums[0]);

    for (int i = 1; i < 35; i++)
    {
        bool found = false;
        vector<double>::iterator iter = orderedNums.begin();
        while (iter != orderedNums.end())
        {
            if (unorderedNums[i] < *iter)
            {
                orderedNums.insert(iter, unorderedNums[i]);
                found = true;
                break;
            }
            iter++;
        }

        if (!found)
            orderedNums.push_back(unorderedNums[i]);
    }
}

bool doFrequencyTest(vector<double> &numArray)
{
    printf("\nPerforming Frequency Test using Kolmogorov-Smirnov method\n");
    printf("\tN = 35; alpha = 0.05\n\n");

    vector<double> orderedNumbers;
    sortNumbers(numArray, orderedNumbers);

    printf("Sorted numbers:\n");
    printNumbers(orderedNumbers);

    // calculate d+ and d- values
    vector<double> dPlus, dMinus;
    for (unsigned int i = 0; i < orderedNumbers.size(); i++)
    {
        int iValue = i + 1;
        double iDivN = (double)iValue / (double)35;
        double dPlusVal = iDivN - orderedNumbers[i];
        double dMinusVal = orderedNumbers[i] - ((double)i / (double)35);

        dPlus.push_back(dPlusVal);
        dMinus.push_back(dMinusVal);
    }

    printf("\nD+ Values:\n");
    printNumbers(dPlus);
    printf("\nD- Values:\n");
    printNumbers(dMinus);

    double criticalDValue = 0.23;
    printf("\nCritical value of D with alpha 0.05, n 35 is 0.23\n" \
           "by table A.8 of Banks, Carlson, Nelson.\n\n");
    bool testPassed = true;
    for (unsigned int i = 0; i < orderedNumbers.size(); i++)
    {
        if ((dPlus[i] > criticalDValue) || (dMinus[i] > criticalDValue))
        {
            testPassed = false;
            printf("FREQUENCY TEST failed for value i=%d\n\n", i);
            return false;
        }
    }
    if (testPassed)
    {
        printf("FREQUENCY TEST Passed for all values\n\n");
        return true;
    }

    return false;
}

bool doRunsUpAndDownTest(vector<double> &numArray)
{
    printf("\nPerforming Runs Up And Down Test\n");
    printf("Numbers again:\n");
    printNumbers(numArray);

    double lastNum = 0;
    char lastChar = ' ';
    char thisChar;
    int runsUp = 0;
    int runsDown = 0;
    vector<double>::iterator iter = numArray.begin();
    while (iter != numArray.end())
    {
        lastNum = *iter;
        iter++;

        if (iter != numArray.end())
        {
            if (*iter > lastNum)
            {
                thisChar = '+';
                if (lastChar != thisChar)
                    runsUp++;
            }
            else if (*iter < lastNum)
            {
                thisChar = '-';
                if (lastChar != thisChar)
                    runsDown++;
            }
            else
            {
                thisChar = ' ';
            }

            printf("%c", thisChar);
            lastChar = thisChar;
        }
    }
    printf("\n");
    printf("Runs Up   - %d\n", runsUp);
    printf("Runs Down - %d\n", runsDown);

    int totalRuns = runsUp + runsDown;
    printf("Total     - %d\n", totalRuns);

    // via formula at the bottom of page 304 of banks, carlson, nelson
    double numPart1 = 2 * numArray.size() - 1;
    double numPart2 = numPart1 / 3.0;
    double num = totalRuns - numPart2;
    double denPart1 = (16 * numArray.size()) - 29;
    double denPart2 = denPart1 / 90.0;
    double den = sqrt(denPart2);
    double z0 = num / den;
    printf("Z0         = %0.9g\n", z0);
    printf("Zsub0.0025 = 1.96\n");
    if ((z0 >= -1.96) && (z0 <= 1.96))
    {
        printf("Z0 is within the acceptable limits (-1.96 to 1.96)\n" \
               "RUNS UP AND DOWN TEST PASSED\n");
        return true;
    }

    printf("RUNS UP AND DOWN TEST FAILED\n");
    return false;
}

bool doRunsAboveAndBelowMeanTest(vector<double> &numArray)
{
```

```
double mean = 0.99999999 / 2.0;
printf("\nPerforming Runs Above and Below Mean Test\n");
printf("Numbers again:\n");
printNumbers(numArray);

printf("Mean is %0.9g\n", mean);

char lastChar = ' ';
char thisChar;
int runsAbove = 0;
int runsBelow = 0;
int numAbove = 0;
int numBelow = 0;
vector<double>::iterator iter = numArray.begin();
while (iter != numArray.end())
{
    if (*iter > mean)
    {
        thisChar = '+';
        numAbove++;
        if (lastChar != thisChar)
            runsAbove++;
    }
    else if (*iter < mean)
    {
        numBelow++;
        thisChar = '-';
        if (lastChar != thisChar)
            runsBelow++;
    }
    else
    {
        thisChar = ' ';
    }

    printf("%c", thisChar);
    lastChar = thisChar;

    iter++;
}
printf("\n");
printf("Runs Above - %d\n", runsAbove);
printf("Runs Below - %d\n", runsBelow);

int totalRuns = runsAbove + runsBelow;
printf("Total Runs - %d\n", totalRuns);

double Nval = numArray.size();

// formula from the bottom of page 306 in banks, carlson, nelson
double numPart1 = (double)(2 * numAbove * numBelow) / Nval;
double num = (double)totalRuns - numPart1 - 0.5;

double denNumPart1 = 2 * numAbove * numBelow;
double denNumPart2 = 2 * numAbove * numBelow - Nval;
double denNum = denNumPart1 * denNumPart2;
double denDen = ((Nval * Nval) * (Nval - 1));
double den = sqrt(denNum / denDen);

double z0 = num / den;
printf("Z0         = %0.9g\n", z0);
printf("Zsub0.0025 = 1.96\n");
```

```
    if ((z0 >= -1.96) && (z0 <= 1.96))
    {
        printf("Z0 is within the acceptable limits (-1.96 to 1.96)\n" \
               "RUNS ABOVE AND BELOW TEST PASSED\n");
        return true;
    }

    printf("RUNS ABOVE AND BELOW TEST FAILED\n");
    return false;
}

double factorial(int value)
{
    if (value == 1)
        return 1;
    return value * factorial(value - 1);
}

bool doRunLengthTest(vector<double> &numArray)
{
    printf("\nPerforming Runs Length Test (using Runs up and down)\n");
    printf("Numbers again:\n");
    printNumbers(numArray);

    double lastNum = 0;
    char lastChar = ' ';
    char thisChar;
    int runsUp = 0;
    int runsDown = 0;

    vector<int> observedRunLengths;
    int currentRunLength = -1;

    // fill the observed lengths with zeroes
    for (unsigned int i = 0; i < numArray.size(); i++)
    {
        observedRunLengths.push_back(0);
    }

    vector<double>::iterator iter = numArray.begin();
    bool firstTime = true;
    while (iter != numArray.end())
    {
        currentRunLength++;
        lastNum = *iter;
        iter++;

        if (iter != numArray.end())
        {
            if (*iter > lastNum)
            {
                thisChar = '+';
                if (lastChar != thisChar)
                    runsUp++;
            }
            else if (*iter < lastNum)
            {
                thisChar = '-';
                if (lastChar != thisChar)
                    runsDown++;
            }
            else
            {
                thisChar = ' ';
            }

            // check run length, but don't do for the first time through
            if ((lastChar != thisChar) && !firstTime)
            {
                observedRunLengths[currentRunLength]++;
                currentRunLength = 0;
            }

            printf("%c", thisChar);
            lastChar = thisChar;
            firstTime = false;
        }
    }
    observedRunLengths[currentRunLength]++;

    printf("\n");
    printf("Runs Up   - %d\n", runsUp);
    printf("Runs Down - %d\n", runsDown);

    int totalRuns = runsUp + runsDown;
    printf("Total     - %d\n", totalRuns);

    double Nval = numArray.size();

    double x0squared = 0;
    for (int j = 1; j < totalRuns; j++)
    {
        double eyj = 0;
        if (j <= Nval - 2)
        {
            double den = factorial(j + 3);
            double numPart1 = (j * j) + (3 * j) + 1;
            double numPart2 = (j * j * j) + (3 * j * j) - j - 4;
            double numPart3 = Nval * numPart1;
            double numPart4 = numPart3 - numPart2;
            double num = 2 * numPart4;

            eyj = num / den;
        }
        else
        {
            double num = 2;
            double den = factorial((int)Nval);
            eyj = num / den;
        }

        double thisIterNumPart1 = observedRunLengths[j] - eyj;
        double thisIterNum = thisIterNumPart1 * thisIterNumPart1;
        double thisIter = thisIterNum / eyj;

        x0squared += thisIter;
    }

    printf("x0 squared = %0.9g\n", x0squared);
    if (x0squared <= 3.84)
    {
        printf("x0 squared is less than the critical value 3.84, " \
               "so RUN LENGTH TEST PASSED\n");
        return true;
    }

    printf("x0 squared is more than the critical value 3.84, " \
           "so RUN LENGTH TEST FAILED\n");
    return false;
}

bool doRunsTest(vector<double> &numArray)
{
    printf("\nPerforming Runs Tests\n");

    bool result = doRunsUpAndDownTest(numArray);
    result = result && doRunsAboveAndBelowMeanTest(numArray);
    result = result && doRunLengthTest(numArray);

    return result;
}

unsigned long getNumberOfGaps(vector<unsigned long> &intArray, unsigned long digit)
{
    vector<unsigned long>::iterator iter = intArray.begin();
    bool hasFoundDigit = false;
    unsigned long gaps = 0;
    while (iter != intArray.end())
    {
        if (*iter == digit)
        {
            if (hasFoundDigit)
                gaps++;
            hasFoundDigit = true;
        }
        iter++;
    }

    return gaps;
}

void getGapLengths(vector<unsigned long> &intArray, unsigned long *lengths)
{
    for (int i = 0; i < 10; i++)
    {
        vector<unsigned long>::iterator iter = intArray.begin();
        bool hasFoundDigit = false;
        unsigned long gaps = 0;
        unsigned long currentGapLength = 0;
        while (iter != intArray.end())
        {
            if (*iter == i)
            {
                if (hasFoundDigit)
                {
                    gaps++;
                    lengths[currentGapLength]++;
                    currentGapLength = 0;
                }
                hasFoundDigit = true;
            }
            else if (hasFoundDigit)
            {
                currentGapLength++;
            }
            iter++;
        }
    }
}

bool doGapTest(vector<unsigned long> &intArray)
{
    printf("\nPerforming Gap Test\n");
    printf("The random integers for the test are:\n");
    printIntegers(intArray);
    int i = 0;

    printf("\nNumber of gaps for each digit, 0 - 9\n");
    for (i = 0; i < 10; i++)
    {
        if (i > 0)
            printf(" ");
        printf("%u", getNumberOfGaps(intArray, i));
    }
    printf("\n");

    unsigned long gapLengths[110];
    memset(gapLengths, 0, sizeof(unsigned long) * 110);
    getGapLengths(intArray, gapLengths);

    printf("\nGap Len    Freq    Rel Freq    Cumul Rel Freq      F(x)    |F(x)-S(x)|\n");
    printf("----------------------------------------------------------------------\n");
    double cumRelFreq = 0;
    double max = 0;
    for (i = 0; i < 100; i+= 4)
    {
        printf("%4d-%2d    ", i, i + 3);
        int freq = 0;
        for (int j = i; j <= i + 3; j++)
        {
            freq += gapLengths[j];
        }
        printf("%4d    ", freq);

        double relFreq = (double)freq / 100.0;
        printf("    %0.2f    ", relFreq);

        cumRelFreq += relFreq;
        printf("          %0.2f    ", cumRelFreq);

        double fx = 1.0 - pow(0.9, i+4);
        printf("%0.4f    ", fx);

        double result = 0;
        if (fx > cumRelFreq)
            result = fx - cumRelFreq;
        else
            result = cumRelFreq - fx;

        printf("     %0.4f", result);

        if (result > max)
            max = result;
        printf("\n");
    }

    double dAlpha = 1.36 / sqrt((double)100);
    printf("\nCritical value of D alpha is %0.9g\n", dAlpha);
    printf("Max D from table is %0.9g\n", max);
    if (max < dAlpha)
    {
        printf("GAP TEST PASSED\n");
        return true;
    }

    printf("GAP TEST FAILED\n");
    return false;
}

bool doPokerTest(vector<unsigned long> &numArray)
{
    printf("\nPerforming Poker Test\n");
    printf("The random numbers for the test are:\n");
    printThreePrecisionNumbers(numArray);

    int expectedNoPairs = (int)(0.72 * numArray.size());
    int expectedThreeAlike = (int)(0.01 * numArray.size());
    int expectedTwoAlike = (int)(0.27 * numArray.size());

    int foundNoPairs = 0;
    int foundThreeAlike = 0;
    int foundTwoAlike = 0;

    for (unsigned int i = 0; i < numArray.size(); i++)
    {
        unsigned long thisValue = numArray.at(i);

        int thirdDigit = (int)(thisValue % 10);
        thisValue /= 10;

        int secondDigit = (int)(thisValue % 10);
        thisValue /= 10;

        int firstDigit = thisValue;

        // check for no like digits
        if ((firstDigit != secondDigit) &&
            (secondDigit != thirdDigit) &&
            (firstDigit != thirdDigit))
        {
            foundNoPairs++;
            continue;
        }

        // check for all the same
        if ((firstDigit == secondDigit) &&
            (firstDigit == thirdDigit))
        {
            foundThreeAlike++;
            continue;
        }

        foundTwoAlike++;
    }

    double noPairsResult = (double)(foundNoPairs - expectedNoPairs);
    noPairsResult *= noPairsResult;
    noPairsResult /= (double)expectedNoPairs;
    double threeAlikeResult = (double)(foundThreeAlike - expectedThreeAlike);
    threeAlikeResult *= threeAlikeResult;
    threeAlikeResult /= (double)expectedThreeAlike;

    double twoAlikeResult = (double)(foundTwoAlike - expectedTwoAlike);
    twoAlikeResult *= twoAlikeResult;
    twoAlikeResult /= (double)expectedTwoAlike;

    double totalResult = noPairsResult + threeAlikeResult + twoAlikeResult;

    printf("\n");
    printf("Combination            Observed (Oi)    Expected (Ei)   ((Oi - Ei)^2)/Ei\n");
    printf("------------------------------------------------------------------------\n");
    printf("Three different digits     %3d              %3d                %0.2f\n",
        foundNoPairs, expectedNoPairs, noPairsResult);
    printf("Three like digits          %3d              %3d                %0.2f\n",
        foundThreeAlike, expectedThreeAlike, threeAlikeResult);
    printf("Three different digits     %3d              %3d                %0.2f\n",
        foundTwoAlike, expectedTwoAlike, twoAlikeResult);
    printf("                           ---              ---                ----------\n");
    printf("                           %3d              %3d                %0.2f\n\n",
        foundNoPairs + foundThreeAlike + foundTwoAlike, expectedNoPairs +
        expectedThreeAlike + expectedTwoAlike, totalResult);

    if (totalResult <= 5.99)
    {
        printf("x0 squared is %0.2f, which less than the critical value 5.99, " \
               "so POKER TEST PASSED\n", totalResult);
        return true;
    }

    printf("x0 squared is %0.2f, which is more than the critical value 5.99, " \
           "so POKER TEST FAILED\n", totalResult);

    return false;
}

bool doSampleAutocorrelationTest(vector<double> &numArray)
{
    printf("\nPerforming Sample Autocorrelation Test\n");
    printf("The random numbers for the test are:\n");
    printTwoPrecisionNumbers(numArray);

    printf("\nTesting autocorrelation of every 5 numbers " \
           "starting at the 3rd number.\n\n");

    int firstI = 3;
    int gap = 5;
    double mDouble = numArray.size() - firstI;
    mDouble /= (double)gap;
    mDouble -= 1;
    int m = (int)mDouble;

    double pHat = 0;
    for (int i = 0; i < m; i++)
    {
        double toAdd = numArray.at(firstI + (i * gap)) *
                       numArray.at(firstI + ((i + 1) * gap));
        pHat += toAdd;
    }

    pHat = ((1.0 / (m + 1)) * pHat) - 0.25;
    double sigmaPHat = (13 * m) + 7;
    sigmaPHat = sqrt(sigmaPHat);
    sigmaPHat /= (12 * (m + 1));

    double z0 = pHat / sigmaPHat;

    double critical = 1.96;

    printf("p-hat = %0.4f\n", pHat);
    printf("sigma p-hat = %0.4f\n", sigmaPHat);
    printf("z0 = p-hat / sigma p-hat = %0.4f\n\n", z0);

    if (abs(z0) < critical)
    {
        printf("The absolute value of z0 is %0.4f, " \
               "which is less than the critical value %0.4f" \
               ", so AUTOCORRELATION SAMPLE TEST PASSED\n", abs(z0), critical);
        return true;
    }

    printf("The absolute value of z0 is %0.4f, " \
           "which is more than the critical value %0.4f" \
           ", so AUTOCORRELATION SAMPLE TEST FAILED\n", abs(z0), critical);

    return false;
}

bool doCompletenessTest()
{
    printf("\nPerforming Completeness Test For 100000 numbers\n");
    bool numbers[100000];
    sgenrand((long)time(NULL));
    initmyrand(100000);

    memset(numbers, 0, 100000 * sizeof(bool));

    bool complete = false;
    unsigned long count = 0;
    while (!complete && (count < 0xFFFFFFFF))
    {
        unsigned long thisNum = myrand();
        numbers[thisNum] = true;
        count++;

        // check to see if they are done
        complete = true;
        for (unsigned long i = 0; i < 100000; i++)
        {
            if (!numbers[i])
            {
                complete = false;
                break;
            }
        }
    }

    if (complete)
        printf("Completeness Test Passed for 100000 numbers " \
               "after %u generated numbers\n", count - 1);
    else
        printf("Completeness Test Failed!\n");
    return complete;
}

void main(int argc, char **argv)
{
    vector<double> numbers;
    vector<unsigned long> threePrecNumbers;
    vector<unsigned long> integers;
    vector<double> twoPrecNumbers;

    generateNumbers(numbers);
    generateIntegers(integers);
    generateTwoPrecisionNumbers(twoPrecNumbers);
    generateThreePrecisionNumbers(threePrecNumbers);

    printNumbers(numbers);

    bool result = doFrequencyTest(numbers);
    if (!result)
    {
        printf("RANDOMNESS TEST FAILED!\n");
        return;
    }

    result = doRunsTest(numbers);
    if (!result)
    {
        printf("RANDOMNESS TEST FAILED!\n");
        return;
    }

    result = doSampleAutocorrelationTest(twoPrecNumbers);
    if (!result)
    {
        printf("RANDOMNESS TEST FAILED!\n");
        return;
    }

    result = doGapTest(integers);
    if (!result)
    {
        printf("RANDOMNESS TEST FAILED!\n");
        return;
    }

    result = doPokerTest(threePrecNumbers);
    if (!result)
    {
        printf("RANDOMNESS TEST FAILED!\n");
        return;
    }

    result = doCompletenessTest();
    if (!result)
    {
        printf("RANDOMNESS TEST FAILED!\n");
        return;
    }

    printf("RANDOMNESS TEST PASSED!\n");
}
```

# Appendix B

# Source Code For An Actor Library

As a reference to use in creating new actor libraries, the full source code for the LimitedConnectionsNodeLibrary actor library is included in this chapter. This code compiles with no errors or warnings under Microsoft Visual Studio .NET 2003.

## B.1 LimitedConnectionsNodeLibrary.h

```
//************************************************************************
// FILE: LimitedConnectionsNodeLibrary.h
//
// AUTHOR: Christopher Brian Shirey
// Florida Institute Of Technology
// Department of Computer Science
//
// CREATED: February 27, 2004
//
// DESCRIPTION: Contains definition for the class LimitedConnectionsNodeLibrary
//************************************************************************
#pragma once

#include "NodeLibrary.h"

using namespace SimulatorEngine;

namespace LimitedConnectionsLibrary
{
    //************************************************************************
    // represents the library for infectable network nodes
    //************************************************************************
    class NodeLibraryApi LimitedConnectionsNodeLibrary : public NodeLibrary
    {
    public:
        LimitedConnectionsNodeLibrary();
        ~LimitedConnectionsNodeLibrary();

        virtual NetworkNode *CreateNode(vector<string> *params, unsigned short numSockets,
            unsigned short timeOut, unsigned char numConnections);
    };
}

//**********************************************************************
// Method:        GetLibraryNetworkNode
// Description: returns an instance of this library
//
// Parameters:
//    None
//
// Return Value: a new instance of this library
//**********************************************************************
NodeLibrary *GetLibraryNetworkNode()
{
    return new LimitedConnectionsLibrary::LimitedConnectionsNodeLibrary();
}
```

## B.2 LimitedConnectionsNodeLibrary.cpp

```
//**********************************************************************
// FILE:        LimitedConnectionsNodeLibrary.cpp
//
// AUTHOR:        Christopher Brian Shirey
//                Florida Institute Of Technology
//                Department of Computer Science
//
// CREATED:        February 27, 2004
//
// DESCRIPTION: Contains implementation for the class LimitedConnectionsNodeLibrary
//**********************************************************************
#define NodeLibrary_File

#include "LimitedConnectionsNodeLibrary.h"
#include "LimitedConnectionsNode.h"

namespace LimitedConnectionsLibrary
{
    //**********************************************************************
    // Method:        LimitedConnectionsNodeLibrary
    // Description: Constructor for the LimitedConnectionsNodeLibrary class
    //
    // Parameters:
    //    None
    //
    // Return Value: None
    //**********************************************************************
    LimitedConnectionsNodeLibrary::LimitedConnectionsNodeLibrary()
    {
    }

    //**********************************************************************
    // Method:        ~LimitedConnectionsNodeLibrary
    // Description: Destructor for the LimitedConnectionsNodeLibrary class
    //
    // Parameters:
    //    None
    //
    // Return Value: None
    //**********************************************************************
    LimitedConnectionsNodeLibrary::~LimitedConnectionsNodeLibrary()
    {
    }

    //**********************************************************************
    // Method:        CreateNode
    // Description: Creates an infectable node and returns it
    //
    // Parameters:
    //    params - a list of param strings to pass to the node
    //    numSockets - the number of sockets for the new node
    //    timeOut - the timeout for the new node
    //    numConnections - the number of connection attempts for the new node
    //
    // Return Value: a new infectable network node
    //**********************************************************************
    NetworkNode *LimitedConnectionsNodeLibrary::CreateNode(vector<string> *params,
        unsigned short numSockets, unsigned short timeOut, unsigned char numConnections)
    {
        LimitedConnectionsNode *node = new LimitedConnectionsNode(params);
        node->SetSockets(numSockets);
        node->SetTimeOut(timeOut);
        node->SetConnectionAttempts(numConnections);
        return node;
    }
}
```

## B.3 LimitedConnectionsNode.h

```
//**********************************************************************
// FILE:        LimitedConnectionsNode.h
//
// AUTHOR:        Christopher Brian Shirey
//                Florida Institute Of Technology
//                Department of Computer Science
//
// CREATED:        February 27, 2004
//
// DESCRIPTION: Contains definition for the class LimitedConnectionsNode
//**********************************************************************
#pragma once

#include "NetworkNode.h"

namespace LimitedConnectionsLibrary
{
    //**********************************************************************
    // represents an infectable network node
    //**********************************************************************
    class NodeLibraryApi LimitedConnectionsNode : public SimulatorEngine::NetworkNode
    {
    public:
        LimitedConnectionsNode(vector<string> *params);
        ~LimitedConnectionsNode();

        virtual bool WillAttemptToConnectToNode(NetworkNode *nodeToConnectTo);
        virtual SimulatorEngine::ConnectionResult ConnectToNode(NetworkNode *connectingNode,
            NetworkNode *nodeToConnect, unsigned short timeSocketHasBeenOpen);
        virtual SimulatorEngine::InfectionResult AttemptToBeInfected(
            NetworkNode *infectingNode);
    protected:
        unsigned int numConnectionsLimitedTo;
    };
}
```

## B.4 LimitedConnectionsNode.cpp

```
//***********************************************************************
// FILE:        LimitedConnectionsNode.cpp
//
// AUTHOR:        Christopher Brian Shirey
//                Florida Institute Of Technology
//                Department of Computer Science
//
// CREATED:        February 27, 2004
//
// DESCRIPTION: Contains implementation for the class LimitedConnectionsNode
//***********************************************************************
#define NodeLibrary_File

#include <windows.h>
#include "LimitedConnectionsNode.h"

namespace LimitedConnectionsLibrary
{
    //***********************************************************************
    // Method:        LimitedConnectionsNode
    // Description: Constructor for the LimitedConnectionsNode class
    //
    // Parameters:
    //    params - a list of parameter strings
    //
    // Return Value: None
    //***********************************************************************
    LimitedConnectionsNode::LimitedConnectionsNode(vector<string> *params)
        : NetworkNode(params)
    {
        infectable = true;
        infected = false;
        type = "Limited Connections Node";
        numConnectionsLimitedTo = 10;

        if (params && (params->size() > 0))
        {
            numConnectionsLimitedTo = atoi(params->at(0).c_str());
        }
    }

    //***********************************************************************
    // Method:        ~LimitedConnectionsNode
    // Description: Destructor for the LimitedConnectionsNode class
    //
    // Parameters:
    //    None
    //
    // Return Value: None
    //***********************************************************************
    LimitedConnectionsNode::~LimitedConnectionsNode()
    {
    }
    //**********************************************************************
    // Method:        WillAttemptToConnectToNode
    // Description: returns whether or not this node will attempt to connect
    //    to the specified node
    //
    // Parameters:
    //    nodeToConnectTo - the node to determine whether or not to connect to
    //
    // Return Value: a connection result enumeration
    //**********************************************************************
    bool LimitedConnectionsNode::WillAttemptToConnectToNode(NetworkNode *nodeToConnectTo)
    {
        if (this->GetNumConnectionsFrom() > numConnectionsLimitedTo)
            return false;

        return true;
    }

    //**********************************************************************
    // Method:        ConnectToNode
    // Description: tries to connect to this node from another node
    //
    // Parameters:
    //    connectingNode - the node trying to connect
    //    nodeToConnect - the node being connected to
    //    timeSocketHasBeenOpen - the number of timesteps the socket has been open
    //        between the 2
    //
    // Return Value: a connection result enumeration
    //**********************************************************************
    SimulatorEngine::ConnectionResult LimitedConnectionsNode::ConnectToNode(
        NetworkNode *connectingNode, NetworkNode *nodeToConnect,
        unsigned short timeSocketHasBeenOpen)
    {
        return SimulatorEngine::Success;
    }

    //**********************************************************************
    // Method:        AttemptToBeInfected
    // Description: tries to be infected by the infectingNode
    //
    // Parameters:
    //    infectingNode - the node trying to infect this one
    //
    // Return Value: a infection result enumeration
    //**********************************************************************
    SimulatorEngine::InfectionResult LimitedConnectionsNode::AttemptToBeInfected(
        NetworkNode *infectingNode)
    {
        return SimulatorEngine::Infected;
    }
}

//**********************************************************************
// Method:        DidNetworkNodeComeFromThisLibrary
// Description: determines whether or not a network node was created by this
//    library
//
// Parameters:
//    node - the node to test
//
```

```
// Return Value: true if this dll created the node, false otherwise
//************************************************************************
bool DidNetworkNodeComeFromThisLibrary(SimulatorEngine::NetworkNode *node)
{
    return (static_cast<LimitedConnectionsLibrary::LimitedConnectionsNode *>
           (node) != NULL);
}
```

## B.5 LimitedConnectionsNodeLibrary.def

```
LIBRARY    LimitedConnectionsNodeLibrary
EXPORTS
GetLibraryNetworkNode
DidNetworkNodeComeFromThisLibrary
```

## B.6 LimitedConnectionsNodeLibrary.vcproj

```
<?xml version="1.0" encoding="Windows-1252"?>
<VisualStudioProject
    ProjectType="Visual C++"
    Version="7.10"
    Name="LimitedConnectionsNodeLibrary"
    ProjectGUID="{39770D7C-9B00-48E3-B862-3F3E8F8005AC}"
    Keyword="Win32Proj">
    <Platforms>
        <Platform
            Name="Win32"/>
    </Platforms>
    <Configurations>
        <Configuration
            Name="Debug|Win32"
            OutputDirectory="Debug"
            IntermediateDirectory="Debug"
            ConfigurationType="2"
            CharacterSet="2">
            <Tool
                Name="VCCLCompilerTool"
                Optimization="0"
                AdditionalIncludeDirectories="..\hephaestusengine"
                PreprocessorDefinitions="WIN32;_DEBUG;_WINDOWS;_USRDLL" \
                                        ";LIMITEDCONNECTIONSNODELIBRARY_EXPORTS"
                MinimalRebuild="TRUE"
                BasicRuntimeChecks="3"
                RuntimeLibrary="2"
                UsePrecompiledHeader="0"
                WarningLevel="3"
                Detect64BitPortabilityProblems="TRUE"
                DebugInformationFormat="4"/>
            <Tool
                Name="VCCustomBuildTool"/>
            <Tool
                Name="VCLinkerTool"
                AdditionalDependencies="hephaestusengine.lib"
                OutputFile="$(OutDir)/LimitedConnectionsNodeLibrary.dll"
                LinkIncremental="2"
                AdditionalLibraryDirectories="..\hephaestusengine\debug"
                ModuleDefinitionFile="LimitedConnectionsNodeLibrary.def"
                GenerateDebugInformation="TRUE"
                ProgramDatabaseFile="$(OutDir)/LimitedConnectionsNodeLibrary.pdb"
                SubSystem="2"
                ImportLibrary="$(OutDir)/LimitedConnectionsNodeLibrary.lib"
                TargetMachine="1"/>
            <Tool
                Name="VCMIDLTool"/>
            <Tool
                Name="VCPostBuildEventTool"
                CommandLine="copy debug\LimitedConnectionsNodelibrary.dll ..\actors"/>
            <Tool
                Name="VCPreBuildEventTool"/>
            <Tool
                Name="VCPreLinkEventTool"/>
            <Tool
                Name="VCResourceCompilerTool"/>
            <Tool
                Name="VCWebServiceProxyGeneratorTool"/>
            <Tool
                Name="VCXMLDataGeneratorTool"/>
            <Tool
                Name="VCWebDeploymentTool"/>
            <Tool
                Name="VCManagedWrapperGeneratorTool"/>
            <Tool
                Name="VCAuxiliaryManagedWrapperGeneratorTool"/>
        </Configuration>
        <Configuration
            Name="Release|Win32"
            OutputDirectory="Release"
            IntermediateDirectory="Release"
            ConfigurationType="2"
            CharacterSet="2">
            <Tool
                Name="VCCLCompilerTool"
                AdditionalIncludeDirectories="..\hephaestusengine"
                PreprocessorDefinitions="WIN32;NDEBUG;_WINDOWS;_USRDLL;
                                         LIMITEDCONNECTIONSNODELIBRARY_EXPORTS"
                RuntimeLibrary="2"
                UsePrecompiledHeader="0"
                WarningLevel="3"
                Detect64BitPortabilityProblems="TRUE"
                DebugInformationFormat="3"/>
            <Tool
                Name="VCCustomBuildTool"/>
            <Tool
                Name="VCLinkerTool"
                AdditionalDependencies="hephaestusengine.lib"
                OutputFile="$(OutDir)/LimitedConnectionsNodeLibrary.dll"
                LinkIncremental="1"
                AdditionalLibraryDirectories="..\hephaestusengine\release"
                ModuleDefinitionFile="LimitedConnectionsNodeLibrary.def"
                GenerateDebugInformation="TRUE"
                SubSystem="2"
                OptimizeReferences="2"
                EnableCOMDATFolding="2"
                ImportLibrary="$(OutDir)/LimitedConnectionsNodeLibrary.lib"
                TargetMachine="1"/>
            <Tool
                Name="VCMIDLTool"/>
            <Tool
                Name="VCPostBuildEventTool"
                CommandLine="copy release\LimitedConnectionsNodelibrary.dll ..\actors"/>
```

```
			<Tool
				Name="VCPreBuildEventTool"/>
			<Tool
				Name="VCPreLinkEventTool"/>
			<Tool
				Name="VCResourceCompilerTool"/>
			<Tool
				Name="VCWebServiceProxyGeneratorTool"/>
			<Tool
				Name="VCXMLDataGeneratorTool"/>
			<Tool
				Name="VCWebDeploymentTool"/>
			<Tool
				Name="VCManagedWrapperGeneratorTool"/>
			<Tool
				Name="VCAuxiliaryManagedWrapperGeneratorTool"/>
		</Configuration>
	</Configurations>
	<References>
	</References>
	<Files>
		<Filter
			Name="Source Files"
			Filter="cpp;c;cxx;def;odl;idl;hpj;bat;asm;asmx"
			UniqueIdentifier="{4FC737F1-C7A5-4376-A066-2A32D752A2FF}">
			<File
				RelativePath=".\LimitedConnectionsNode.cpp">
			</File>
			<File
				RelativePath=".\LimitedConnectionsNodeLibrary.cpp">
			</File>
			<File
				RelativePath=".\LimitedConnectionsNodeLibrary.def">
			</File>
		</Filter>
		<Filter
			Name="Header Files"
			Filter="h;hpp;hxx;hm;inl;inc;xsd"
			UniqueIdentifier="{93995380-89BD-4b04-88EB-625FBE52EBFB}">
			<File
				RelativePath=".\LimitedConnectionsNode.h">
			</File>
			<File
				RelativePath=".\LimitedConnectionsNodeLibrary.h">
			</File>
		</Filter>
		<Filter
			Name="Resource Files"
			Filter="rc;ico;cur;bmp;dlg;rc2;rct;bin;rgs;gif;jpg;jpeg;jpe;resx"
			UniqueIdentifier="{67DA6AB6-F800-4c08-8B7A-83BB121AAD01}">
		</Filter>
	</Files>
	<Globals>
	</Globals>
</VisualStudioProject>
```

# Bibliography

[1] C. Kerényi. *The Gods of the Greeks*, page 71. Thames and Hudson, 1951.

[2] Virus related statistics. `http://www.securitystats.com/virusstats.html`.

[3] Virus history. `http://www.cknow.com/vtutor/vthistory.htm`.

[4] The cost of 'code red': $1.2 billion. `http://www.usatoday.com/tech/news/2001-08-01-code-red-costs.htm`.

[5] Ian Whalley, Bill Arnold, David Chess, John Morar, Alla Segal, and Morton Swimmer. An environment for controlled worm replication and analysis. `http://researchweb.watson.ibm.com/antivirus/SciPapers/VB2000INW.pdf`.

[6] Giuseppe Serazzi and Stefano Zanero. Computer virus propagation models. `http://www.elet.polimi.it/upload/zanero/papers/zanero-serazzi-virus.pdf`.

[7] Fred Cohen. Computer viruses - theory and experiments. `http://vx.netlux.org/lib/afc01.html`.

[8] Jeffrey O. Kephart and Steve R. White. Directed-graph epidemiological models of computer viruses. `http://researchweb.watson.ibm.com/antivirus/SciPapers/Kephart/VIRIEEE/virieee.gopher.html`.

[9] Winfried Gleissner. A mathematical theory for the spread of computer viruses, 1989.

[10] Peter S. Tippett. Computer virus replication, 1990.

[11] Alan Solomon. Epidemiology and computer viruses. `http://ftp.cerias.purdue.edu/pub/doc/viruses/epidemiology_and_viruses.txt`.

[12] Yang Wang and Chenxi Wang. Modeling the effects of timing parameters on virus propagation. `http://www.ece.cmu.edu/~chenxi/pub/worm.pdf`.

[13] Jeffrey O. Kephart and Steve R. White. Measuring and modeling computer virus prevalence. http://www.research.ibm.com/antivirus/SciPapers/Kephart/PREV/prevalence.gopher.html.

[14] Matthew M. Williamson and Jasmin Léveillé. An epidemiological model of virus spread and cleanup. www.hpl.hp.com/techreports/2003/HPL-2003-39.pdf.

[15] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to own the internet in your spare time. http://www.csd.uch.gr/~hy558/reports/jkapad-present2.ppt.

[16] Zesheng Chen, Lixin Gao, and Kevin Kwait. Modeling the spread of active worms. www.ieee-infocom.org/2003/papers/46_03.PDF.

[17] Steve R. White. Open problems in computer virus research. Presented at Virus Bulletin Conference, Munich, Germany, October 1998. http://researchweb.watson.ibm.com/antivirus/SciPapers/White/Problems/Problems.html.

[18] What is monte carlo simulation? http://www.decisioneering.com/monte-carlo-simulation.html.

[19] Nicholas C. Weaver. Warhol worms: The potential for very fast internet plagues. http://www.cs.berkeley.edu/~nweaver/warhol.html.

[20] Bruce Ediger. Simulating network worms. http://www.users.qwest.net/~eballen1/nws/.

[21] Ddosvax project. http://www.tik.ee.ethz.ch/~ddosvax/.

[22] Scalable simulation framework. http://www.ssfnet.org.

[23] Ssf.app.worm: A network worm modeling package for ssfnet. http://www.cs.dartmouth.edu/~mili/research/ssf/worm/.

[24] Michael Liljenstam, David M. Nicol, Vincent H. Berk, and Robert S. Gray. Simulating realistic network worm traffic for worm warning system design and testing. Presented in WORM '03 in Washington, D.C. on October 27, 2003.

[25] Dan Geer, Rebecca Bace, Peter Gutmann, Perry Metzger, Charles P. Pfleeger, John S. Quarterman, and Bruce Schneier. Cyberinsecurity: The cost of monopoly. how the dominance of microsoft's products poses a risk to security. http://www.ccianet.org/papers/cyberinsecurity.pdf.

[26] Richard Ford. Microsoft, monopolies and migraines: The role of monoculture. http://www.virusbtn.com/magazine/archives/200312/monoculture.xml.

[27] Tom Liston. Welcome to my tarpit: The tactical and strategic use of labrea. http://www.google.com/search?q=cache:LnRKzqhIygQJ:sunsite.ccu.edu.tw/pub12/sourceforge/l/labrea/LaBrea-Tom-Liston-Whitepaper-Welcome-to-my-tarpit.txt+liston+Welcome+To+My+Tarpit&hl=en. Original article posted on Hackbusters.net is no longer available.

[28] Jerry Banks, John S. Carson, and Barry L. Nelson. *Discrete-Event System Simulation*, chapter 8. Prentice Hall, 1996.

[29] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/mt.pdf.

[30] Mersenne primes: History, theorems and lists. http://www.utm.edu/research/primes/mersenne/.

[31] Linear feedback shift register. http://www.fact-index.com/l/li/linear_feedback_shift_register.html.

[32] Mersenne twister. http://home.ecn.ab.ca/~jsavard/crypto/co4814.htm.

[33] Kolmogorov-smirnov test. http://www.itl.nist.gov/div898/handbook/prc/section2/prc212.htm.

[34] Jerry Banks, John S. Carson, and Barry L. Nelson. *Discrete-Event System Simulation*, page 539. Prentice Hall, 1996.

[35] Chi-square goodness-of-fit test. http://www.itl.nist.gov/div898/handbook/prc/section2/prc211.htm.