San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

12-1-2010

Improved Worm Simulator and Simulations

Arnold Suvatne
San Jose State University

Follow this and additional works at: http://scholarworks.sjsu.edu/etd_projects

Recommended Citation
Suvatne, Arnold, "Improved Worm Simulator and Simulations" (2010). *Master's Projects.* Paper 10.

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

IMPROVED WORM SIMULATOR AND SIMULATIONS

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Arnold Suvatne

November 29, 2010

SAN JOSÉ STATE UNIVERSITY

The Undersigned Thesis Committee Approves the Writing Project Titled

IMPROVED WORM SIMULATOR AND SIMULATIONS

by
Arnold Suvatne

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

_____

Dr. Mark Stamp, Department of Computer Science Date

_____

Dr. Chris Pollett, Department of Computer Science Date

_____

Dr. Teng Moh, Department of Computer Science Date

APPROVED FOR THE UNIVERSITY

_____

Associate Dean Office of Graduate Studies and Research Date

Abstract

According to the latest Microsoft Security Intelligence Report (SIR), worms were the second most prevalent information security threat detected in the first half of 2010 – the top threat being Trojans. Given the prevalence and damaging effects of worms, research and development of worm counter strategies are garnering an increased level of attention. However, it is extremely risky to test and observe worm spread behavior on a public network. What is needed is a packet level worm simulator that would allow researchers to develop and test counter strategies against rapidly spreading worms in a controlled and isolated environment. Jyotsna Krishnaswamy, a recent SJSU graduate student, successfully implemented a packet level worm simulator called the Wormulator. The Wormulator was specifically designed to simulate the behavior of the SQL Slammer worm. This project aims to improve the Wormulator by addressing some of its limitations. The resulting implementation will be called the Improved Worm Simulator.

Table of Contents

1 Introduction 1

2 Computer Worms 3

2.1 Worm Propagation Methods 4

2.2 SQL Slammer Worm 6

3 Wormulator 8

3.1 Wormulator Architecture and Functional Overview 9

3.2 Wormulator Implementation 12

3.3 Opportunities for Improvement 15

4 Improved Worm Simulator 17

4.1 Improved Worm Simulator Architecture and Functional Overview 17

4.2 Improved Worm Simulator Implementation 19

5 Simulation Setup and Results 29

5.1 Simulation Setup 29

5.2 Simulation Results and Analysis 31

5.3 Performance 39

6 Future Work 40

7 Conclusion 41

References 42

LIST OF FIGURES

Figure 3.1-1 System Architecture of Wormulator 10
Figure 3.2-1 Pseudo-code for the LNS in Wormulator [1] 13
Figure 3.2-2 Pseudo-code of a Simulated Network Node [1] 14
Figure 3.2-3 Pseudo-code of the UDP Worm [1] 14
Figure 4.1-1 System Architecture of Improved Worm Simulator 18
Figure 4.2-1 Pseudo-code for the SCU (Simulation Control Unit) 23
Figure 4.2-2 Pseudo-code for processPackets 24
Figure 4.2-3 Pseudo-code for a Temporary Worm Thread 24
Figure 4.2-4 Pseudo-code for a Persistent Worm Thread 25
Figure 5.1-1 start_iws script 30
Figure 5.2-1 Simulation output 32
Figure 5.2-2 Worm Traffic Rate – 1M Hosts, 100K Vulnerable, 2M Bandwidth 33
Figure 5.2-3 Infected Node Count – 1M Hosts, 100K Vulnerable, 2M Bandwidth 34
Figure 5.2-4 Worm Traffic Rate – 2M Hosts, 100K Vulnerable, 3M Bandwidth 35
Figure 5.2-5 Infected Node Count – 2M Hosts, 100K Vulnerable, 3M Bandwidth 36
Figure 5.2-6 Worm Traffic Rate – 5M Hosts, 100K Vulnerable, 3M Bandwidth 37
Figure 5.2-7 Infected Node Count – 5M Hosts, 100K Vulnerable, 3M Bandwidth 38

LIST OF EQUATIONS

Equation 4.2-1 Bandwidth threshold for an iteration 27
Equation 4.2-2 Worm Traffic Rate 28

1 Introduction

The goal of this project is to improve an existing worm simulator called the Wormulator. The resulting implementation will be called the Improved Worm Simulator.

Computer worms are the second most prevalent information security threat found by anti-malware software today – the most prevalent being Trojans [15]. Given the growth of the Internet and the sheer volume of critical information and money that is exchanged daily through the Internet, a fast spreading worm could inflict tremendous financial losses to individuals, businesses, and other organizations across the globe. The effects of recent real world computer worms have already shown us the financial devastation that could result. For example, the financial loss caused by the Conficker worm had been estimated to be as high as $9.1 billion [18].

Not only do worms put a tremendous amount of money at risk, but people's safety and personal security can be at risk as well. Communication systems, safety and security services, and emergency response systems are becoming increasingly reliant on networked resources that are accessed via the Internet. If any of those systems are disrupted by massive congestion or failures in the network, people's lives could be at stake. For example, the Slammer worm in 2003 generated enough network congestion to cause failures in the 911-emergency system in Bellevue, Washington [4].

The potential devastating effects of future computer worms has many information security experts, especially those affiliated with large enterprises, interested in worm research. Consequently, there's a considerable amount of money, time and effort being invested towards

the study of worm behavior models, worm detection methods, automated worm containment mechanisms, worm traffic simulations, etc [7]. Unfortunately, there are some significant challenges to studying worms.

Due to the disruptive nature and potential damaging effects of worms, it is highly unethical to perform worm experiments on the Internet, regardless of intentions. The disruption and damages caused by even an accidental release of a worm can draw severe penalties, including criminal prosecution. Without the luxury of being able to use the Internet as their testing ground, worm researchers are forced to resort to running simulations in a relatively small isolated network.

Researchers have taken one of two approaches to worm simulation. One approach is to simulate worms by using mathematical models that leverage epidemiological equations used to describe the spread of real-world diseases [13]. Another approach is to use packet level worm simulations that leverage existing network simulators [17]. A recent SJSU graduate project took a lightweight network simulator known as the Spamulator (used to simulate spam propagation) and enhanced it to simulate the SQL Slammer worm. The Spamulator enhancement is called the Wormulator [1].

The purpose of this project is to make the following improvements to the Wormulator:

1. Increase the scalability of the Wormulator from a few thousand active worm nodes to at least one hundred thousand active worm nodes.

2. Improve the Wormulator's simulation of the Slammer worm by implementing a more realistic handling of network saturation, as opposed to the automatic gradual slowdown to a near standstill, as exhibited by Wormulator.

3. Implement a more realistic infection probability – the Wormulator currently implements a 100% infection probability.

The remainder of this paper is organized as follows:

- **Section 2 Computer Worms**: defines what computer worms are, explains how computer worms function, describes various worm propagation methods, and provides an overview of the SQL Slammer worm.
- **Section 3 Wormulator**: describes the Wormulator architecture, gives a functional overview of the Wormulator, provides the implementation details of the Wormulator, and highlights some improvement opportunities for the Wormulator.
- **Section 4 Improved Worm Simulator**: details the architecture, functional overview and implementation of the Improved Worm Simulator.
- **Section 5 Simulation Setup and Results**: describes the simulation setup, presents the simulation results, and discusses the simulator's performance.
- **Section 6 Future Work**: discusses deficiencies that were not addressed by the Improved Worm Simulator and proposes future work.
- **Section 7**: **Conclusion**

2 Computer Worms

Simply put, computer worms are self-propagating malware. Malware is defined as any software that has malicious intent, hence the prefix "mal." The key difference between worms

and viruses is that worms actively search for and infect vulnerable machines across a network [22]. Once the initial computer is infected, a worm uses the network to spread itself from one computer to another by exploiting network security vulnerabilities in operating systems and/or applications on other computers. Each newly infected node will attempt to infect all other nodes that it can reach. As the number of infected nodes increase, the number of network wide infection attempts (scans) increase. And as the number of network wide infection attempts increase, the number of newly infected nodes increases at a much faster rate, and so on. Leveraging the Internet's volume of potential victims and highly connected infrastructure has proven to be a highly effective method for propagating worms.

2.1 Worm Propagation Methods

We can characterize a worm by the method in which it propagates itself. Some of the key components of a worm's propagation method are:

- Transfer mechanism
- The manner in which it selects target addresses
- Speed

A worm will typically use one or more transfer mechanisms to copy itself to other computers. A worm may use message exchange applications such as email, Internet relay chat, or instant messaging. A worm might use a network protocol such as TCP or UDP. Or a worm might use inter-process communication mechanisms such as named pipes, shared memory, or shared folders to transfer itself to other computers.

One of the primary functions of a worm is to seek out potential victims to infect. But, in order to infect another computer, the worm must know the target computer's IP address, and the target computer must be vulnerable to the worm's attack. There are two general ways for worms to choose their targets. A worm can randomly generate target addresses, or a worm can use a pre-generated list of known vulnerable target addresses.

If a worm randomly generates the target addresses, it can use a uniform random scanning method, or a local preference scanning method. In the uniform random scanning method, a worm randomly and uniformly selects IP addresses from the entire range of Internet addresses. Uniform random scanning is highly inefficient, considering there are only a relatively small number of vulnerable hosts in such a large address space as the entire Internet [14] [16]. In local preference scanning, a worm randomly selects IP addresses, but the selection method is biased towards addresses that are located in local networks, such as enterprise intranets [17]. Local preference scanning worms tend to spread much more rapidly. There are two reasons for this. Number one, the addresses from a local network address space tend to have a higher probability of being reachable and active. Number two, the density of vulnerable hosts tend to be higher in a local network, because hosts in a local network tend to be of the same type and tend to be configured in the same way.

Although a worm that uses local preference scanning can be much more efficient than a worm that uses uniform random scanning, they pale in comparison to a worm that uses a pre-generated "hit list" of known vulnerable target addresses. Such a worm is referred to as a **flash worm**, and can be orders of magnitudes faster. Given an accurate list of known vulnerable host addresses, the predicted speed at which a **flash worm** can infect vulnerable hosts is incredible. It has been shown that, given pristine conditions, a **flash worm** can theoretically infect over 95%

of a million vulnerable hosts in just over half a second [6]. However, the key to a **flash worm**'s success is an accurate list of vulnerable host addresses. Due to the highly complex and dynamic nature of the Internet, it can be quite a monumental task to procure a reliable list of valid and vulnerable addresses. It is for this reason that the feasibility of **flash worm**s is questioned by most experts. However, it would be unwise to ignore the possibility of an effective **flash worm**. Researchers have proposed methods to circumvent problems that arise from the highly dynamic nature of the Internet [6].

Speed is another component of a worm's propagation method. One might assume that a worm must scan as fast as possible in order to infect as many victims as possible before it is detected and removed. Interestingly enough, a slow scanning worm has a much higher chance of infecting more computers than a fast scanning worm [16]. A slow scanning worm can evade detection because the traffic that it generates appears to be normal. A fast scanning worm, on the other hand, can generate an unusually high number of connection attempts which can easily be detected and removed from an infected computer, thus limiting the spread of the worm.

Regardless of the specific methods used to propagate a worm, the fact that worms can propagate on their own, have global reach, and can spread at rapid rates, makes worms one of the most effective and devastating forms of malware out there. The next subsection gives an overview of the SQL Slammer worm. The SQL Slammer worm is the real-world worm that the Improved Worm Simulator attempts to model.

2.2 SQL Slammer Worm

Of the twenty-three worms that were released in the "wild" this past decade, SQL Slammer is one that has garnered a tremendous amount of attention and notoriety [12]. This is due to the speed that Slammer exhibited, as well as the significant damage that it inflicted, in terms of financial and productivity loss.

SQL Slammer, or just Slammer, was unleashed upon the Internet on January 25, 2003 [4]. As of the writing of this report, Slammer has the distinction of being the fastest worm in history. Slammer achieved its remarkable speed by simply sending 404 byte UDP packets that exploited buffer overflow vulnerabilities in Microsoft's SQL Server software [9]. By using UDP, Slammer did not have to wait for a response from its target – as would have been the case, if TCP had been used. Instead, Slammer could continually send UDP packets as fast as the host's network bandwidth would allow. Given a 100 Mbps network connection, Slammer could manage a scan rate of around 26,000 scans per second [4]. Ironically, it was Slammer's aggressive nature and incredible speed that limited its success.

Slammer used the uniform random scanning method to select its targets. By using the uniform random scanning method, Slammer achieved an initial exponential growth, doubling the infected population every 8.5 seconds [19]. Within a few minutes, however, the infection rate slowed significantly due to continual retries of already infected hosts, non-existing hosts, and immune hosts. In addition to that, the scan traffic overwhelmed the global network causing numerous network outages and system failures, which further exacerbated the slowing of the infection rate [4].

Unfortunately, by that time, Slammer had already done significant damage. In South Korea, twenty-seven million people lost their cell phone and Internet access. In Portugal, three hundred

thousand people had their cable modems crash on them. Corporate email systems around the globe slowed to a grinding halt. 911 emergency response teams had to resort to the use of pencil and paper to handle emergency calls. Airlines had to cancel many flights due to their inability to process tickets [19].

Although worms typically carry a malicious payload, Slammer showed us that worms do not necessarily need to have malicious payloads to cause considerable harm. All told, the cost of Slammer had been estimated to be between $950 million and $1.2 billion [20].

3 Wormulator

The simulator implemented in this project is built upon a previous implementation of a worm simulator called the Wormulator. This section gives some background on the Wormulator, describes the Wormulator's architecture, gives a functional overview of the Wormulator, provides some implementation details of the Wormulator, and describes some opportunities for improvement.

The Wormulator is a packet level worm simulation program developed by Jyotsna Krishnaswamy, a former Graduate student of San Jose State University [1]. It is an extension of the Spamulator, a lightweight network simulator that runs on a single machine developed by John Aycock, Heather Crawford, and Rennie deGraaf to teach a course on spam and spyware [3]. The main objective of the Wormulator was to simulate the behavior of the SQL Slammer Worm using a virtual network of thousands of server nodes running on a single machine [1]. But, in order to simulate the SQL Slammer worm, the handling of User Datagram Protocol (UDP)

packets had to be supported. Unfortunately, the Spamulator only supported the Transmission Control Protocol (TCP), thus the Spamulator had to be modified to support UDP. In addition to providing support for UDP, the final version of Wormulator included these additional enhancements [1]:

- Simulated network nodes are executed as threads, instead of independent processes – the original implementation of Wormulator used independent processes, which limited the scale of Wormulator to only a couple thousand nodes [1].
- Network traffic measurements – the number of packets in the network, measured once per second of simulation time.
- Simulation of congestion in the network – signals congestion when the network traffic increases beyond a pre-defined bandwidth limit, and throttles the traffic rate accordingly.

3.1 Wormulator Architecture and Functional Overview

Depicted in Figure 3.1-1 below is the system architecture of the Wormulator. The large box that encompasses everything but the "Outside World" represents a single computer, be it a laptop, PC, or server. At the core of the Wormulator is the Loopback Network Simulator (LNS) [1]. The LNS creates and initializes all simulated network nodes, handles all of the incoming worm traffic, puts newly infected network nodes into worm propagation mode, keeps track of the traffic rate, and simulates network congestion.

Initially, when the Wormulator is first launched, there are no simulated network nodes running. A setup script called *servers.pl* is executed to create the simulated network nodes. The number of simulated network nodes desired is configurable and defined in the setup script. For

each simulated network node to be simulated a “wake up” packet is sent to the Wormulator. For each “wake-up” packet received by the Wormulator, a new thread is created and that represents a distinct simulated network node. Every simulated network node starts out in the uninfected state. They simply run in a loop listening for UDP packets.

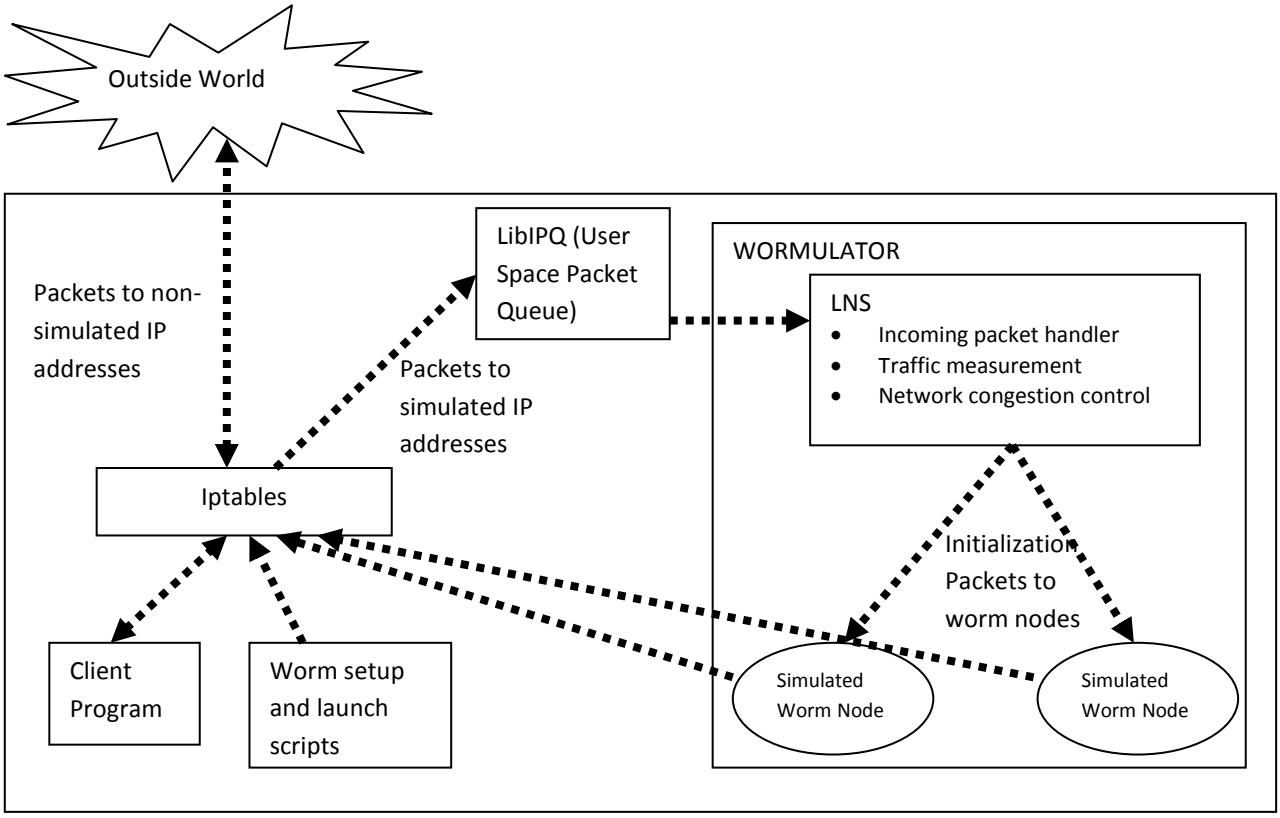

Figure 3.1-1 System Architecture of Wormulator

To actually begin the worm spread, the *worm_spread.pl* script is executed. The *worm_spread.pl* script sends a single UDP packet to infect one of the simulated network nodes. That initial infected node then begins randomly generating target IP addresses and sends worm packets to each of the generated target addresses. If a randomly generated target address matches the address of a simulated network node then a successful infection will be achieved.

Each newly infected worm node in turn randomly generates target IP addresses and sends worm packets to each the generated target address. Each infected worm node continues sending worm packets indefinitely, or until the Wormulator is killed by the user.

A critical requirement when simulating a worm is to ensure that the worm traffic does not escape to the "outside world." Unfortunately, the Wormulator must use the network interface card of the local machine to send and receive packets [1]. Therefore, the network card must be active and connected to the network while executing the tool. So to ensure that all worm traffic remains on the local machine, iptables and the libIPQ library are used to redirect worm traffic back to the Wormulator. Packets destined for a simulated worm node are redirected by IP Table rules to a user space packet queue called the IPQ, implemented in the libIPQ library.

Every packet that ends up on the IPQ ultimately gets handled by the packet handler routine in the Wormulator's LNS module [1]. The LNS module retrieves packets from the IPQ and checks the destination address of each packet to determine if a new network node gets infected or not. If the packet is destined for an uninfected node then that node is marked as infected and gets put into worm propagation mode. If the packet is destined for an already infected node then the LNS module simply marks the packet as accepted and releases it back to the iptables for final traversal. Packets destined for non-simulated IP addresses are allowed to proceed to its destination, untouched by the Wormulator.

3.2 Wormulator Implementation

Wormulator leverages the bulk of its implementation from the Linux version of the Spamulator. Apart from the server initialization and worm launch scripts, all of the modifications were done in a single source code module – the lns.cpp module [1]. The core functionality of Wormulator lies primarily in the LNS module. The pseudo-code for the Wormulator's LNS module is given in Figure 3.2-1. The LNS module begins by creating a user datagram socket and listens for incoming UDP packets. When the LNS module receives a UDP packet, the destination address and port of the packet is determined.

If the destination address and port does not match that of an already infected worm node then a new thread is created, and that new thread executes the worm code described in Figure 3.2-2. The newly created worm node thread creates a user datagram socket, sets the 'server alive' flag to true, and waits for the next packet. When a second packet is received, the worm node is considered infected, and the worm propagation routine is called. The pseudo-code for worm propagation is given in Figure 3.2-3. The worm is a random scanning worm that models the behavior of the SQL Slammer worm [1].

The worm generates a random IP address from the pool of simulated worm node addresses and sends a UDP packet to that address. It repeats this indefinitely. If the destination address and port does match that of an already infected worm node, then the LNS module updates the network traffic count, checks for congestion, adjusts the worm scan rate appropriately, marks the packet as accepted, and releases the packet for final traversal through the IP Table rules.

```
Define parameters:
        ▪  networkTraffic = 0.
        ▪  NETWORK_BANDWIDTH = 10000.
        ▪  congestion = FALSE;

Create User Datagram socket

Receive packet P
   ▪  P is from source IP address As port Ps
   ▪  P is to destination IP address Ad, port Pd

If in the same instance of time
           networkTraffic ++
else:
           networkTraffic = 0

if P is a new connection:
      create a new thread with unique thread Id, tId:
         ▪  Invoke the server
      store (Ad, Pd, P'd) in table T
      store tId in a tIdList
      rewrite P into P:
         ▪  change As to 127.0.0.1
         ▪  change Ad to 127.0.0.1
         ▪  change Pd to P'd
else:
      find entry (Ad, Pd,?) in T
      rewrite P into P':
         ▪  change As to 127.0.0.1
         ▪  change Ad to 127.0.0.1
         ▪  change Pd to P'd found in T
         ▪
If (networkTraffic > NETWORK_BANDWIDTH)
      Congestion = TRUE

If (Congestion)
      send signal SIGUSR2 to all servers in tIdList
else:
      send signal SIGUSR1 to all servers in tIdList

send P'
```

Figure 3.2-1 Pseudo-code for the LNS in Wormulator [1]

```
Create UDP socket and bind to a unique port number P

Write port number P to ostream, which is read by Wormulator

Create file to log statistics

Set up signal handlers:
   ▪ SIGINT cleans up & terminates server executable
   ▪ SIGUSR2 indicates congestion free network.
   ▪ SIGUSR1 indicates congested network with no bandwidth.

Receive worm packet P
   ▪ P is from source IP address 127.0.0.1 port 1
   ▪ P is to destination IP address 127.0.0.1 port P

if P is an empty packet:
      do nothing, as this packet is sent to bring the server alive.

if P is a first non empty packet:
      create a pthread to handle the worm packet.

Record time of infection in the log file.
```

Figure 3.2-2 Pseudo-code of a Simulated Network Node [1]

```
Define the following:
      wormMessage = absolute path of the worm
      sleepRange = 100.0 ms

Create UDP socket to send self

In an infinite loop:
   ▪ generate random IP address of a simulated server, As
   ▪ calculate the corresponding port number, Ps
   ▪ send the 'wormMessage' to As: Ps
   ▪ sleep for the duration of 'sleepRange'
```

Figure 3.2-3 Pseudo-code of the UDP Worm [1]

The network traffic count is accumulated between time ticks – each tick marks the passing of one second of clock time. On every tick of the clock the network traffic count of the previous time period is logged, and the network traffic count is reset to zero. The signals SIGUSR1 and SIGUSR2 are user defined signals used to simulate the effect of congestion in the network. The SIGUSR2 signal is used to indicate a congestion free network. From the start of the simulation until the point at which the network saturation threshold is reached, the SIGUSR2 signal is signaled at every time tick. Each time the SIGUSR2 signal is signaled, the sleepRange is decreased by a certain amount, which in turn increases the rate at which each worm node sends packets. Once the bandwidth saturation threshold is reached, congestion in the network is indicated by the SIGUSR1 signal and the sleepRange is decreased at every new instant of time.

3.3 Opportunities for Improvement

This section describes three ways that the Wormulator can be improved. Although, there may be many other opportunities for improvement, only the ones that will be addressed by the Improved Worm Simulator are discussed here.

As mentioned earlier, the original implementation of the Wormulator used independent Linux processes to represent each simulated network node. The overhead of each independent process, in terms of memory utilization and slow context switching, severely limited the scale of the original Wormulator. The original Wormulator was only able to scale to a few thousand simulated worm nodes. An attempt was made to improve the scalability of the Wormulator by converting to the use of POSIX threads instead of independent processes to represent each worm

node. Unfortunately, just simply converting to the use of POSIX threads did not resolve the scalability issue. Even by running the thread based version of the simulator on a dual-core processor machine, the simulation still had difficulties scaling beyond a few thousand nodes. With thousands of threads running simultaneously, both processors quickly reached maximum utilization, and the host machine quickly became unresponsive. There is clearly an opportunity for improvement here.

Another opportunity for improvement is in the manner in which Wormulator simulates the behavior of the worm once the network bandwidth saturation threshold is reached. Upon reaching the network bandwidth saturation threshold, Wormulator automatically and precipitously throttles down the rate at which each worm node sends worm packets. Recall that this is achieved by increasing the sleepRange at every time tick. Unfortunately, this throttling down continues indefinitely, resulting in a precipitous decline in the network traffic, and before long the simulation slows to a virtual standstill. A more accurate representation of actual worm behavior is for the network traffic to level out at the saturation point.

The final opportunity for improvement is in how Wormulator handles infection probability. Wormulator assumes that every randomly generated address is a vulnerable address. In other words, every worm packet sent by a worm node has a 100% success rate of infecting its target, assuming that the target is not already infected. This is inaccurate. First, the target address may not even be a valid address. Second, the host that's associated with the target address may be disconnected or inactive. And finally, the host associated with the target address may not have the specific software or configuration that is vulnerable to the worm attack.

4 Improved Worm Simulator

The Improved Worm Simulator is a Linux based packet level simulator that improves upon the Wormulator described in Section 3. This section describes the architecture, functional overview, and implementation of the Improved Worm Simulator.

4.1 Improved Worm Simulator Architecture and Functional Overview

The first improvement objective for the Improved Worm Simulator was to scale the simulation to at least 100,000 active worm nodes. One potential solution would have been to implement a distributed simulation using multiple physical interconnected machines, similar to the distributed worm simulation design in [17]. However, due to limited resources and funding, this was not feasible for this project. One of the beneficial features of both the Spamulator and the Wormulator was their ability run on a single physical machine. The authors of the Spamulator indicated that this was an important goal, since a great number of researchers have limited resources and funding [3]. Since the Improved Worm Simulator project was likewise limited in resources and funding, it was decided that the Improved Worm Simulator would be designed to run on a single machine as well. In order to accomplish this, some fundamental changes had to be made to the architecture and functionality of the original simulator.

The system architecture of the Improved Worm Simulator is depicted in Figure 4.1-1. Once again, the large box that encompasses everything but the “Outside World” represents a single computer, be it a laptop, PC, or server. The bulk of the Improved Worm Simulator’s functionality is contained in a central control unit called the Simulation Control Unit (SCU).

The SCU's role is analogous to the Wormulator's LNS module, but the functionality differs substantially. A brief summary of the functionality and responsibilities of the SCU is provided in the box representing the SCU, in Figure 4.1-1.

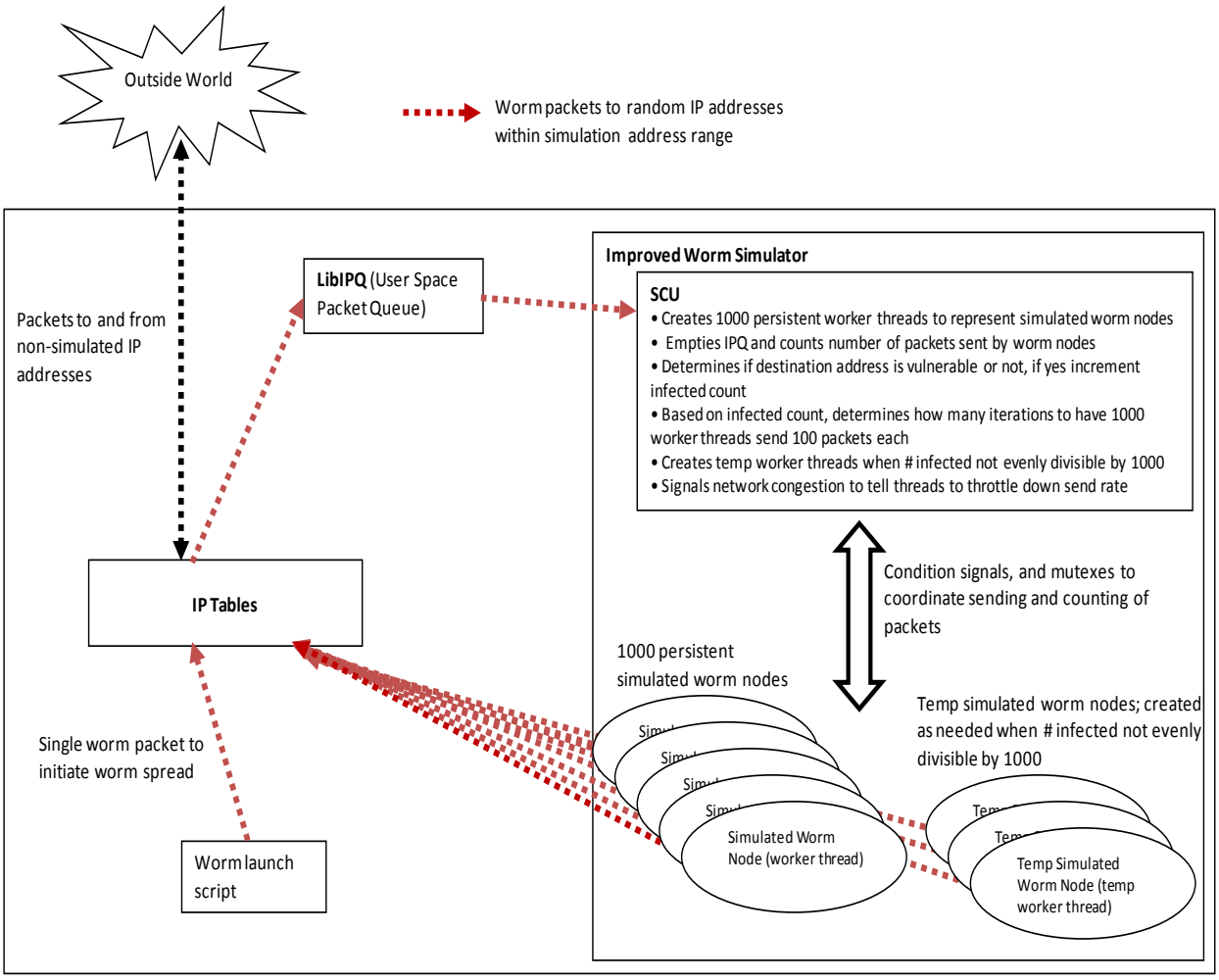

Figure 4.1-1 System Architecture of Improved Worm Simulator

Also depicted in the architectural diagram, Figure 4.1-1, are a set of 1,000 persistent simulated worm node threads and a set of temporary simulated worm node threads. The

difference between a persistent worm node thread and a temporary one will be explained later in Section 4.2, the implementation section. These persistent and temporary simulated worm node threads represent network nodes that have been infected. Their primary role is to send worm packets to randomly generated target addresses. Red dotted arrows are used to show the path of the worm packets through the system.

A key addition to the system is a set of thread based condition signals and mutual exclusion control variables. These condition signals and mutual exclusion control variables are critical to the complex thread synchronization needs of the Improved Worm Simulator. The implementation section will describe how these condition signals and mutual exclusion variables are used.

The iptables and the libIPQ library are used in the exact same way as it was in the Wormulator – to redirect worm traffic back to the simulator, thereby preventing worm packets from reaching the "outside world." The worm setup script, *servers.pl*, is no longer needed. All of the setup and configuration is done by the simulation startup script, *start_iws*, or hardcoded in the Improved Worm Simulator.

4.2 Improved Worm Simulator Implementation

Just as the Wormulator leveraged the bulk of the Spamulator for its implementation, the Improved Worm Simulator leverages the bulk of the Wormulator for its implementation. A majority of the changes were made to the core module – the lns.cpp module. Some changes were made to the configuration and setup as well.

Recall that having more than a few thousand threads running concurrently on a single machine was prohibitive. Despite the fact that the Linux machine used for this project was equipped with a dual-core processor, both memory and processor resources were quickly exhausted, rendering the simulation and the Linux machine unresponsive. Through experimentation, it was discovered that running 1,000 concurrent threads was manageable. Given this constraint, the scalability problem was reduced to: how can we simulate 100,000 active worm nodes running a maximum of 1,000 concurrent threads at any given time? The solution: implement a cycle based simulation, where every simulated active worm node is allowed to complete a specified set of activities at each time cycle before moving on to the next time cycle.

In the Improved Worm Simulator, each simulated active worm node sends 100 UDP packets during each time cycle – each packet destined to a randomly generated address. This means that all simulated active worm nodes would be allotted time to send 100 packets before the simulation moves on to the next time cycle. If this is done in a sequential fashion with 100,000 active worm nodes, the simulation timescale factor would be 100,000 to 1. In other words, it would take 100,000 seconds, or 27 hours and 46 minutes, to simulate 1 second of real time. Obviously, this timescale factor is not practical.

To reduce the timescale factor down to a more reasonable 100 to 1, the Improved Worm Simulator takes advantage of concurrent threads. During each time cycle, the Improved Worm Simulator simulates active worm nodes in groups of up to 1,000 concurrent threads at a time until all active worm nodes have been represented for the current cycle. For example, if there are 3,500 active worm nodes, then in a single time cycle there would be 3 iterations running 1,000

concurrent threads and 1 iteration running 500 concurrent threads. This essentially simulates having all 3,500 active worm nodes concurrently sending 100 packets each.

The Improved Worm Simulator is designed to scale to any number of active worm nodes. For example, you could scale the simulation to 1,000,000 active worm nodes. However, the time scale factor would then be 1,000 to 1 – a little over 17 ½ minutes to simulate 1 second of real time. This is much better than the 100,000 to 1 timescale factor, but it would take a little over 83 hours to simulate 5 minutes of worm activity. The default configuration for the Improved Worm Simulator scales the simulation to 100,000 active worm nodes; that works out to be 100 iterations of running 1,000 concurrent threads during each time cycle – a time scale factor of 100 to 1. At a timescale factor of 100 to 1, it would take a little over 8 hours to simulate 5 minutes of traffic – not ideal, but feasible. This timescale factor is certainly reasonable for simulating fast spreading worms, since you would typically only have to simulate less than a minute of real time.

Note, that scaling to 100,000 active worm nodes does not mean that there are only 100,000 possible addresses in the target address range. It simply means that only 100,000 of the possible target addresses are vulnerable to the worm attack. The number of possible addresses that can be randomly generated is dependent on how many possible unique addresses there are in the simulation's configured target address range. For example, if the configured target address range is any address between 10.1.0.0 to 10.20.200.250, then the number of possible target addresses is (20 x 200 x 250) = 1,000,000. Given this address range and a 10% infection probability, the simulator can simulate 1,000,000 network nodes, with 100,000 nodes being vulnerable to infection. Incidentally, the infection probability is one of the improvements made in the Improved Worm Simulator. However, instead of aiming for a particular probability percentage,

the code is adjusted to maintain 100,000 maximum infections. The infection probability is therefore dependent on the size of the target address range. If the size of the target address range is 2,000,000 then the infection probability is 5%. If the size of the target address range is 5,000,000 then the infection probability is 2%.

When the Improved Worm Simulator is started, the SCU immediately creates 1,000 persistent worm threads. This is done to avoid having to create 1,000 worm threads multiple times during each time cycle. These 1,000 worm threads will persist throughout the entire simulation. Each of these 1,000 persistent worm threads begins execution by waiting on a signal to send packets. The worm spread at this point has not begun yet.

To begin the worm spread, the *worm_spread.pl* script is run. The worm spread script sends a single UDP worm packet destined for address 10.0.0.0. This packet gets redirected to the IPQ by the IP Table rules. The SCU retrieves this initial packet from the IPQ and sets the infected node count to 1. The SCU then creates a temporary worm thread. This temporary worm thread sends 100 UDP worm packets to randomly generated addresses selected from the configured target address range, and then exits. Pseudo-code for the SCU, the SCU's packet processing routine, the temporary worm thread, and the persistent worm thread are given in Figures 4.2-1, 4.2-2, 4.2-3, and 4.2-4 respectively.

```
Define Global Variables:
  MAXTHREADS = 1000, InfectedNodeCount = 0
  WormTrafficRate = 0, maxPackets = 0

Create MAXTHREADS # of persistent worm threads

Wait for initial packet to arrive in IPQ
Get initial packet P from IPQ
   ▪  P destination is address Ad, port Pd

InfectedNodeCount = 1

Loop Forever:
  IterationCount = InfectedNodeCount / MAXTHREADS
  TempCount = InfectedNodeCount mod MAXTHREADS

  For 1 to IterationCount: // run 1,000 persistent worm threads/iteration
    Record send packets start time
    Set sendPackets condition to 1
    Broadcast sendPackets condition
    Wait on condition to process packets
    Call processPackets
  End For

  maxPackets = TempCount * PACKETS_TO_SEND

  If TempCount > 0:
    Record send packets start time
  End If

  For 1 to TempCount: // run temp worm threads
    Create temp worm thread
  End For

  If TempCount > 0:
    Wait on condition to process packets
    Call processPackets
  End If

  Calculate WormTrafficRate
  Output WormTrafficRate and InfectedNodeCount
  Reset WormTrafficRate
End Loop
```

Figure 4.2-1 Pseudo-code for the SCU (Simulation Control Unit)

```
Set processPackets condition to 0

Loop until IPQ empty:

  Get packet P from IPQ
   ▪  P destination is address Ad, port Pd

  If Ad of packet P is vulnerable and not already infected:
    Store (Ad, Pd, P'd) in infected list L
    InfectedNodeCount = InfectedNodecount + 1
  End If

  Mark P accepted
  Release P

End Loop
```

Figure 4.2-2 Pseudo-code for processPackets

```
Global Variables:
  PACKETS_TO_SEND = 100, SLEEPRANGE = 1000
  maxPackets = TempCount * PACKETS_TO_SEND

Create UDP socket

For 1 to PACKETS_TO_SEND
  Generate random target address Ad, and port Pd from specified range
  Send packet P to target address
  Sleep SLEEPRANGE microseconds
End For

packetsSent = packetsSent + PACKETS_TO_SEND

If packetsSent = maxPackets
  Record send packets end time
  Set processPackets condition to 1
  Signal processPackets condition
End If
```

Figure 4.2-3 Pseudo-code for a Temporary Worm Thread

```
Global Variables:
  PACKETS_TO_SEND = 100, SLEEPRANGE = 1000
  MAXPACKETS = MAXTHREADS * PACKETS_TO_SEND
  threadsDone = 1


Loop Forever:
  Wait on condition to send packets
  Create UDP socket

  For 1 to PACKETS_TO_SEND:
    Generate random target address A_d, and port P_d from specified range
    Send packet P to target address
    Sleep SLEEPRANGE microseconds
  End For

  Close UDP socket

  packetsSent = packetsSent + PACKETS_TO_SEND

  If packetsSent = MAXPACKETS:
    Record send packets end time
    Set sendPackets condition to 0
    Set sendPacketsDone condition to 1
    Broadcast sendPacketsDone condition
    Continue to top of loop
  End If

  Wait on condition sendPacketsDone
  threadsDone = threadsDone + 1

  If threadsDone = MAXTHREADS:
    Set threadsDone = 1
    Set sendPacketsDone condition to 0
    Set processPackets condition to 1
    Signal processPackets condition
  End If

End Loop
```

Figure 4.2-4 Pseudo-code for a Persistent Worm Thread

The 100 UDP worm packets sent by the initial worm node are redirected to the IPQ by the IP Table rules. The SCU retrieves each packet from the IPQ and determines if the destination

address is vulnerable to the infection. If yes, then the address is added to the infected address list, the infected node count is incremented by 1, the packet is marked as accepted, and it is released to traverse the rest of the IP Table rules. If no, the address is not added to the infected address list, the infected node count remains the same, the packet is marked as accepted, and it is released to traverse the rest of the IP Table rules.

After all of the incoming worm packets are processed, it is likely that the infected node count has increased. The number of infected nodes will remain under 1,000 in the early time cycles. While the infected node count remains under 1,000, the SCU creates and executes as many temporary worm threads as there are infected nodes. Once the infected node count exceeds 1,000, then the persistent worm threads are used for every increment of 1000 infected nodes. For example, if the total number of infected nodes is 26,860, then during each time cycle there would be 26 iterations where 1,000 persistent worm threads will send 100 packets each. That accounts for 26,000 infected nodes. If the total number of infected nodes is not equally divisible by 1,000, then temporary worm threads are created to represent the excess number beyond the last increment of 1,000. In this case, there are 860 infected nodes that remain, so during each time cycle an additional 860 temporary worm threads are created and run concurrently. Those 860 temporary worm threads will send 100 packets each and exit.

Recall that in the Wormulator, the LNS module was responsible for handling all incoming worm packets, examining each packet to determine if new nodes have been infected, keeping track of the worm traffic rate, and simulating network congestion. It must do all of this while running concurrently with all of the other worm threads. This meant that the LNS module was competing with thousands of threads for processing time. As a result, the LNS module was running only a fraction of one percent of the time. With so little time to perform its

responsibilities, the LNS could not possibly monitor the worm traffic rate accurately, nor could it remove the packets from the IPQ quick enough to avoid overflow. To resolve this issue, the Improved Worm Simulator alternates the sending and receiving of packets on a per iteration basis.

At the start of each iteration, all of the worm packets for that iteration are sent. While the packets are being sent, the SCU waits on a condition signal. After the last packets are sent, the worm thread that sent the last packet sends a signal to the SCU to wake up and process all of the packets in the IPQ. Upon receiving the process packets signal, the SCU wakes up and begins processing the packets in the IPQ. It examines each packet and determines if any new nodes are infected based on the destination address of each sent packet.

The SCU also determines the current iteration's bandwidth threshold, which is the cut-off point for accepting new packets sent during the iteration. The bandwidth threshold for an iteration is calculated using the following equation:

$$\frac{NETWORK_BANDWIDTH\ in\ packets\ per\ second}{\#\ iterations\ in\ current\ cycle}\ x\ \frac{duration\ of\ current\ iteration\ in\ milliseconds}{1{,}000}$$

Equation 4.2-1 Bandwidth threshold for an iteration

The first part of the equation determines how many packets per second are allotted to each iteration in the time cycle. To determine the *number* of packets allotted for a particular iteration, you have to multiply the packets per second rate by the number of seconds it took to complete the iteration. For example, if the NETWORK_BANDWIDTH is 2,000,000 packets per second,

and the number of iterations for the current cycle is 100, and the duration for the current iteration is 2500 milliseconds (2.5 seconds), then the iteration's bandwidth threshold is 50,000 packets. Implementing congestion control at the iteration level distributes the bandwidth limitation across all active worm nodes. This approach achieves a more realistic behavior than cutting a time cycle short once the bandwidth is exceeded.

After all of the packets are processed, the SCU then broadcasts a signal to inform all of the persistent worm threads to send the next batch of packets, thus beginning the next iteration. If needed, the last iteration might be handled by temporary worm threads, as explained earlier. The completion of the last iteration marks the end of the time cycle. At the end of each time cycle, the following metrics are recorded by the SCU:

- Worm Traffic Rate
- Elapsed simulated real time
- Infected Node Count

The worm traffic rate is calculated using the following equation:

$$\frac{total\ number\ of\ packets\ sent\ during\ time\ cycle}{avg.time\ to\ send\ packets\ in\ an\ iteration\ (milliseconds)}\ x\ \frac{1000\ milliseconds}{second}$$

Equation 4.2-2 Worm Traffic Rate

Note that the denominator is not the total simulation time to complete a time cycle, rather it's the average of all the iteration times during a time cycle. This is because all iterations in a time

cycle are considered to be running parallel to one another, along the same time line. For example, if there are a hundred iterations in a time cycle, and each iteration takes 2 seconds to complete, then the total simulation time to complete the time cycle is 200 seconds, but the elapsed simulated real time for the time cycle is only 2 seconds. The elapsed simulated real times will be used as the horizontal axis values in our simulation results given in the next section. The infected node counts are recorded at the end of each time cycle for the same reason that the worm traffic rates are recorded at the end of each time cycle.

5 Simulation Setup and Results

This section describes the simulation setup and presents some simulation results along with some analysis.

5.1 Simulation Setup

The current version of the Improved Worm Simulator was successfully built and ran on an Intel based PC running the Ubuntu Linux operating system, version 10.10. To build and install the Improved Worm Simulator, the source code should be unzipped to a user home directory (e.g. /home/arnold) and then steps 1 through 3 of the INSTALL file should be followed. There will be some build errors on the first few build attempts. A couple of the errors have to do with some missing libraries, as explained in the INSTALL file. Other build errors result from having invalid directory paths hardcoded in the lns.cpp source file.

Steps 4 and 5 of the INSTALL file and some other system configurations are handled by the start_iws script. The start_iws script is shown in Figure 5.1-1. Line 3 increases the maximum number of threads that can be created in the system. Lines 4-8 increase the sizes of

the ip_queue, socket receive buffer, and socket send buffer. The default size settings are not sufficient to handle the number of packets that are sent and received during each iteration. Line 21 clears any previous iptables rules. Line 24 sets an iptables rule that redirects all UDP packets destined for addresses between 10.0.0.0 and 10.255.255.255 to a user space queue – in this case, the IPQ. With this address range, the simulation could simulate up to 16,581,375 potential target addresses. Incidentally, this specific range was chosen because addresses with 10 as the first octet are local private addresses, thus reducing the chance that worm packets actually reach a real public address outside the private network. Line 25 sets an iptables rule that redirects UDP packets destined for the localhost's loopback address to the IPQ. Finally, line 29 executes the simulator.

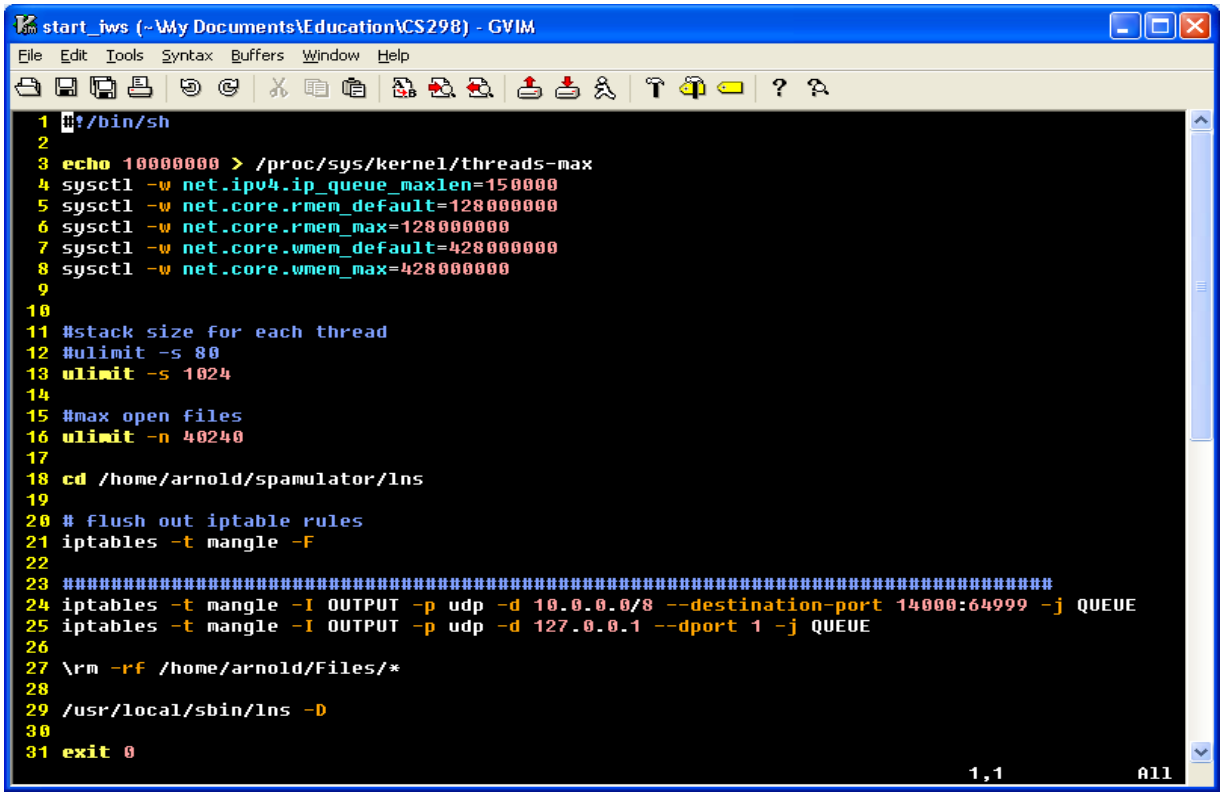

Figure 5.1-1 start_iws script

The following are configurable simulation settings that involve modifying the lns.cpp source file:

- NETWORK_BANDWIDTH – maximum simulation bandwidth in packets per second.
- RANGE2, RANGE3, RANGE4 – max values for the second, third, and fourth octets respectively in the randomly generated target addresses.
- LOWPORT, HIGHPORT – establishes the vulnerable port range, a kludge way to implement infection probability.
- MAXTHREADS – number of persistent worm threads to create at startup.
- PACKETS_TO_SEND – number of packets that each worm thread sends during each time cycle.
- SLEEP_RANGE – number of microseconds to sleep between packets. This is to simulate some sense of path delay.

For the simulator to run properly, the network interface card on the host machine must be enabled and connected. This is perfectly safe, since the worm packets never get beyond the network interface card – they are looped back. To start the simulator, the start_iws script must be run. At this point, the simulator outputs a few messages and then waits for the initial worm packet. To actually begin the worm spread the worm_spread.pl script must be run. CTRL-C is used to terminate the simulation.

5.2 Simulation Results and Analysis

Before the results are presented, the output of the simulations must be explained. Figure 5.2-1 shows a sample of the output generated by a simulation run. Lines with 5 comma separated values represent the worm traffic and timing metrics for a particular iteration. The first

value is the current cycle number. The second value is the current iteration number and the total iterations for the current cycle, separated by a colon. The third value is the elapsed "real" time for that iteration, measured in milliseconds. The fourth value is the total number of packets sent during that iteration. The last value is the total number of packets that were accepted during that iteration. If the last value is less than the fourth value, this indicates that the bandwidth threshold was exceeded and congestion control measures were taken. Each line that starts with "Infected Node Count" represents the total number of nodes that have been infected thus far.

```
.
.
12, 70:74, 2546, 100000, 100000
12, 71:74, 2467, 100000, 98681
12, 72:74, 2486, 100000, 99441
12, 73:74, 2470, 100000, 98801
12, 74:74, 1075, 25300, 25300

Infected Node Count 93786

13, 1:94, 2479, 100000, 76850
13, 2:94, 2442, 100000, 75703
13, 3:94, 2457, 100000, 76168
13, 4:94, 2424, 100000, 75145
13, 5:94, 2427, 100000, 75238
13, 6:94, 2485, 100000, 77036
13, 7:94, 2501, 100000, 77532
13, 8:94, 2406, 100000, 74587
13, 9:94, 2518, 100000, 78059
.
.
```

Figure 5.2-1 Simulation output

To facilitate the charting of the results, the simulation output is also collected in two files located in the /home/user/Files directory. One file is named traffic.csv, which contains the traffic metrics. The other file is named inc.txt, which contains the infected node count data.

To demonstrate the Improved Worm Simulator's capability to correctly model the behavior of a uniform random scanning worm like the SQL Slammer worm, the results of three different simulation runs are presented here. Each simulation is run with a different network size and bandwidth limit combination. The number of vulnerable hosts is approximately the same in each simulation – around 100,000.

Two charts are associated with each simulation run. The first chart plots the worm traffic rate in packets per second against the elapsed time in seconds. The second chart plots the infected node count against the elapsed time in seconds. There are a total of 6 charts, 2 for each simulation.

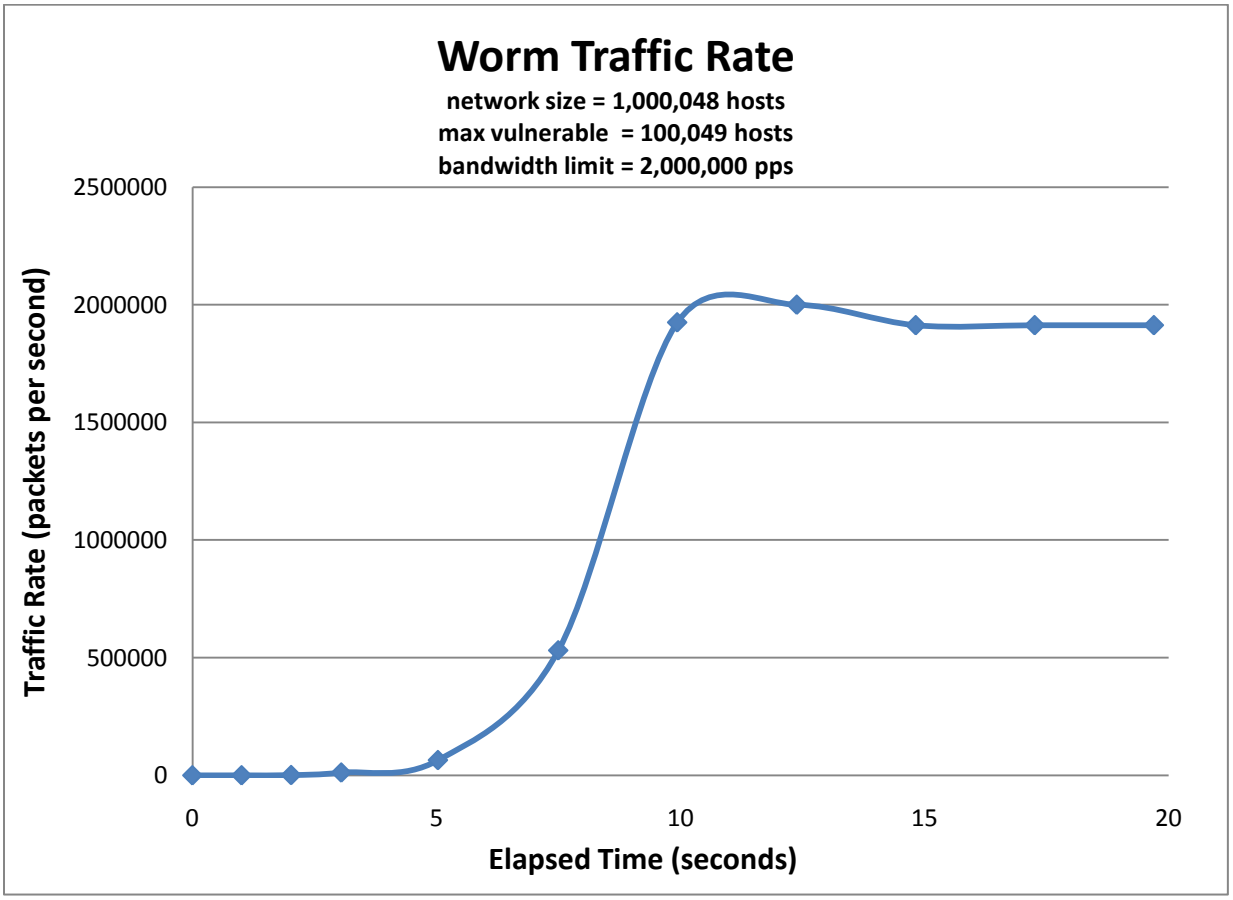

Figure 5.2-2 Worm Traffic Rate – 1M Hosts, 100K Vulnerable, 2M Bandwidth

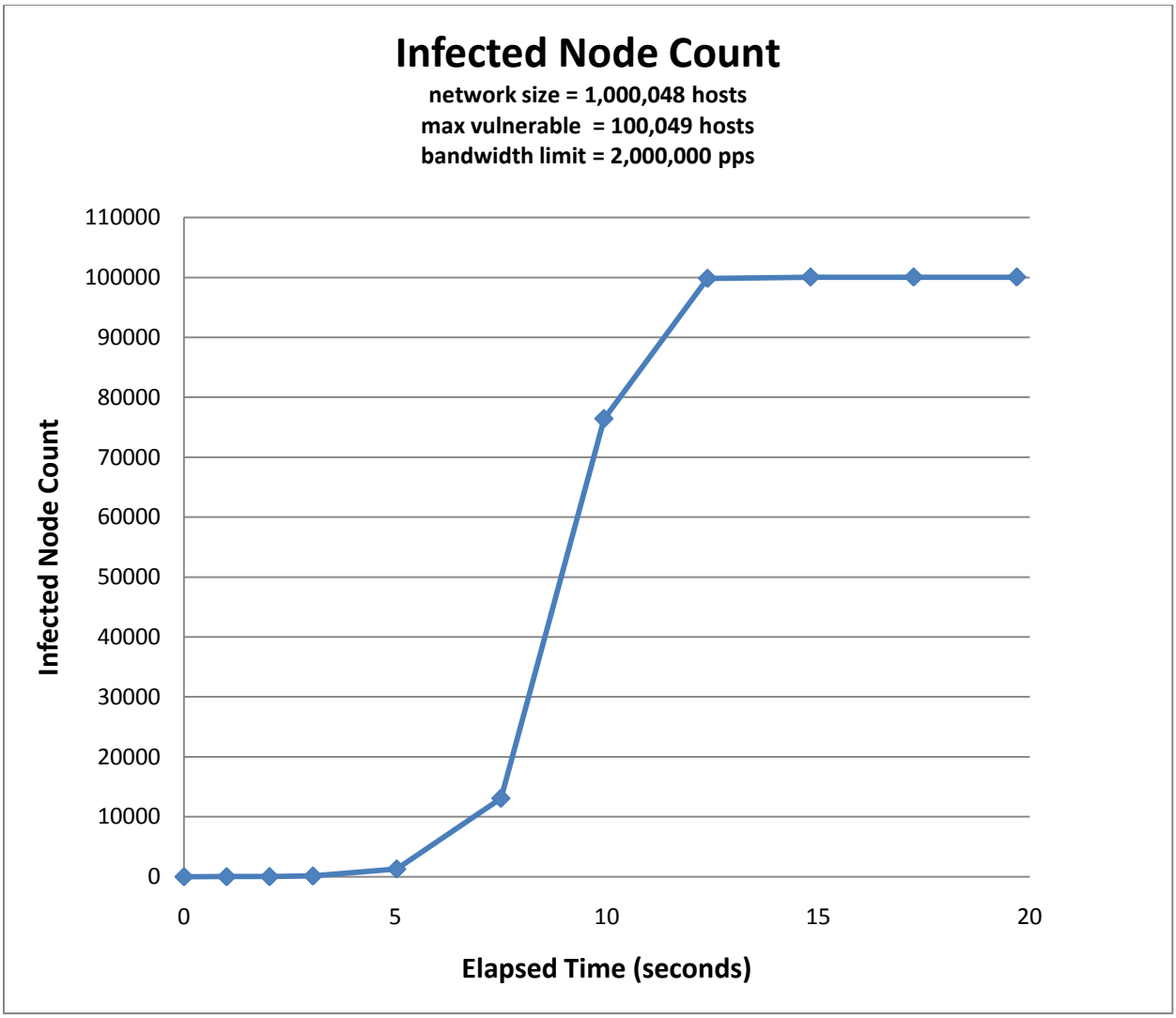

Figure 5.2-3 Infected Node Count – 1M Hosts, 100K Vulnerable, 2M Bandwidth

The results of the first simulation run are shown in Figures 5.2-2 and 5.2-3. This first simulation is configured with a network size of 1 million potential target hosts, with approximately 100,000 of them being vulnerable to the attack. The bandwidth limit for this simulation is set at 2 million packets per second, which is equivalent to what a 7200-NPE-G2 Cisco router can handle [21].

Both the worm traffic rate and the infected node count plots exhibit the classic logistic form of the SQL Slammer worm, where the rate stays relatively low in the early stages and then grows exponentially for a period of time before leveling off at the bandwidth limit.

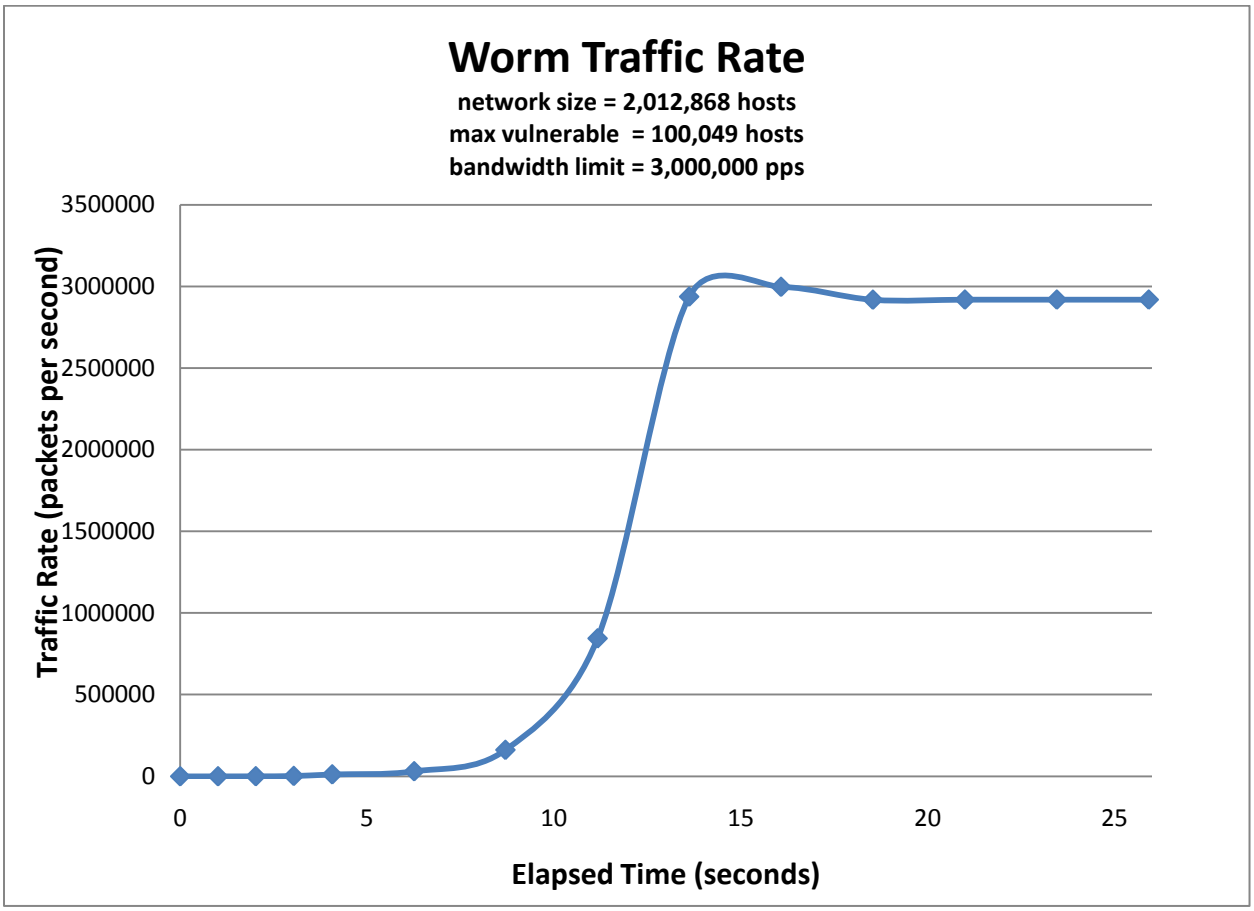

Figure 5.2-4 Worm Traffic Rate – 2M Hosts, 100K Vulnerable, 3M Bandwidth

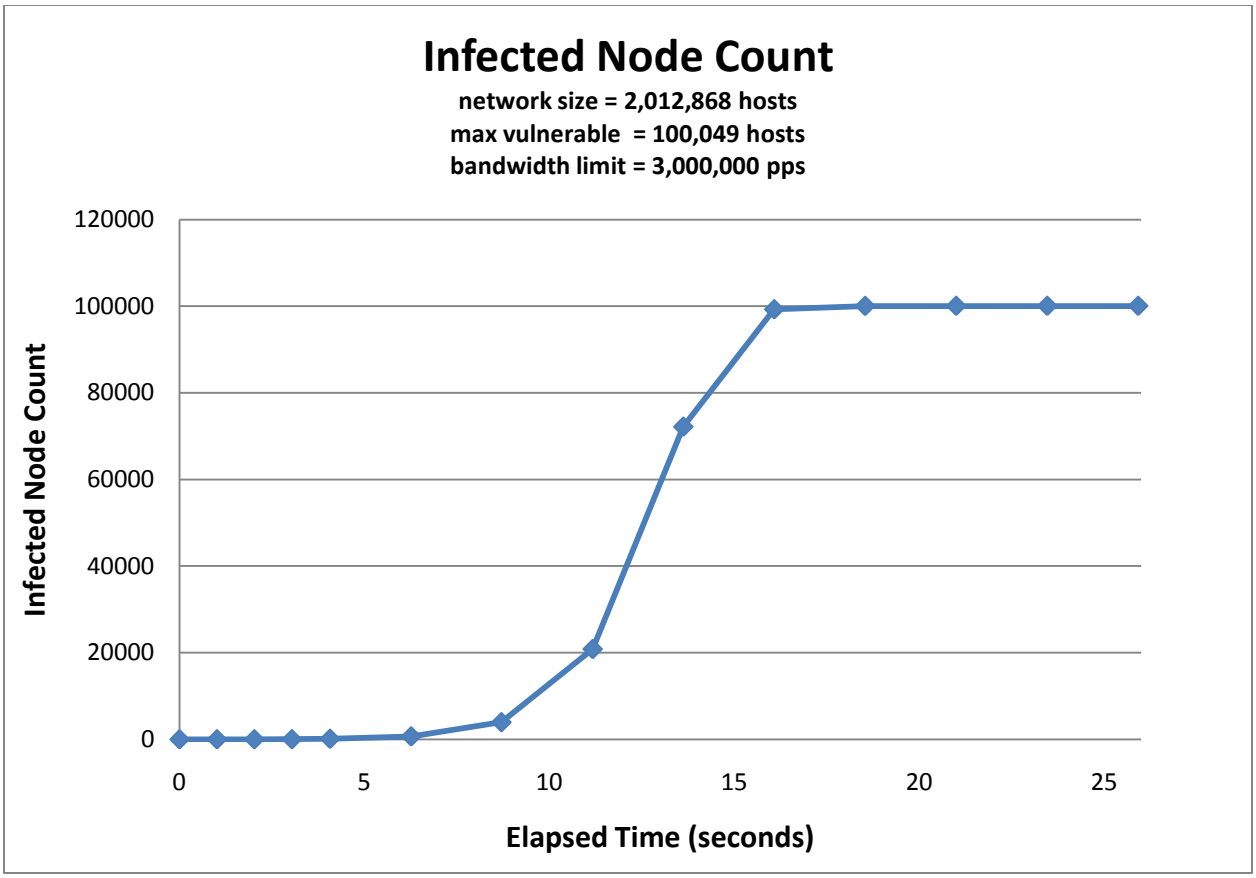

Figure 5.2-5 Infected Node Count – 2M Hosts, 100K Vulnerable, 3M Bandwidth

The second simulation's results are shown in Figures 5.2-4 and 5.2-5. For this simulation the network size is increased to a little over 2 million potential target hosts, and the bandwidth limit is increased to 3 million packets per second. With the network size doubled, the infection probability is essentially halved. This results in slower growth in both the worm traffic rate and infected node count. The slower growth in the worm traffic rate is hard to distinguish because of the scale of the vertical axis in Figure 5.2-4. The difference can really be seen in the infected node count growth. In the first simulation, at close to the 12 second mark, the infected node count is already at its max of 100,000. Whereas in the second simulation, the infected node count is only 45,000 at around the 12 second mark.

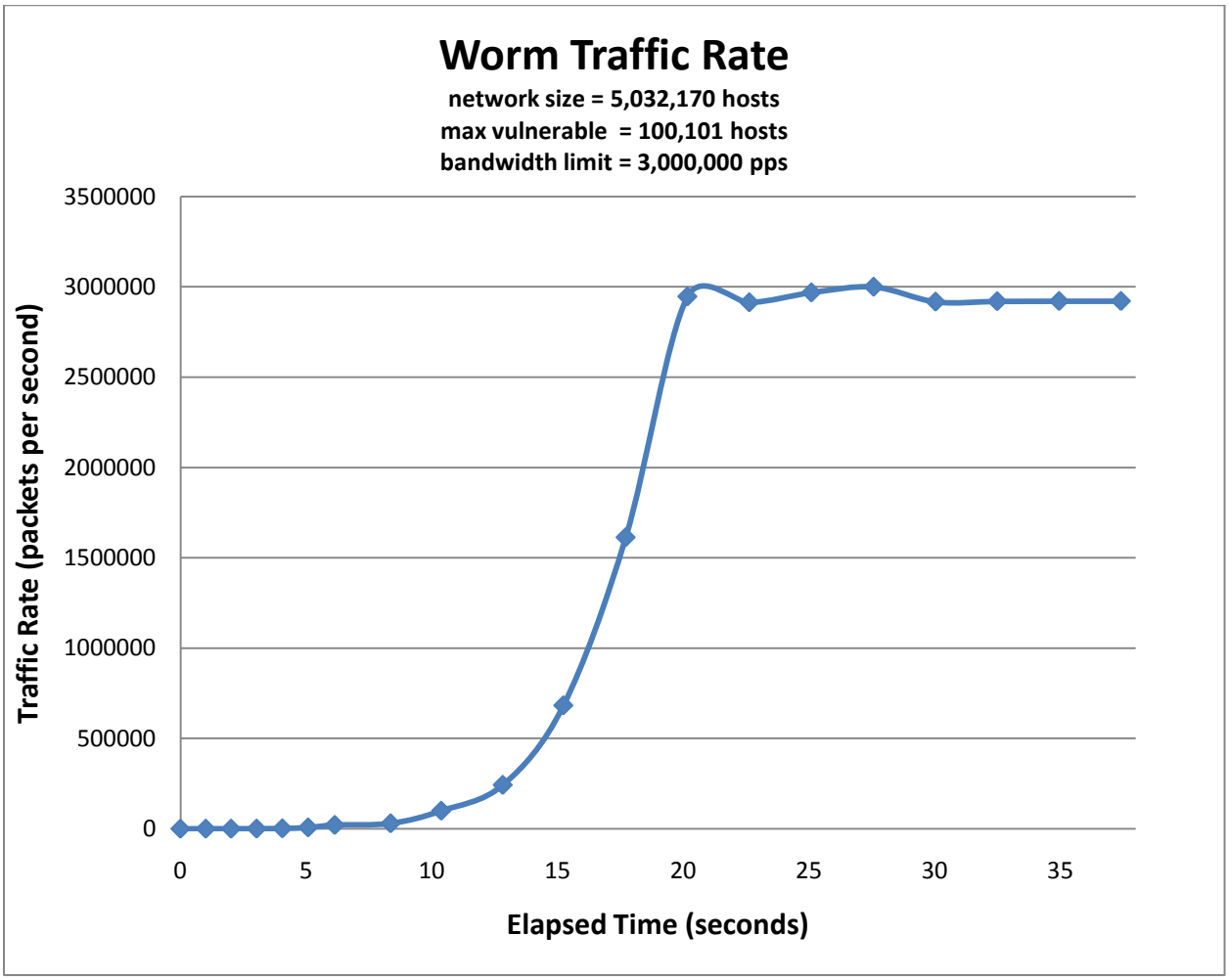

Figure 5.2-6 Worm Traffic Rate – 5M Hosts, 100K Vulnerable, 3M Bandwidth

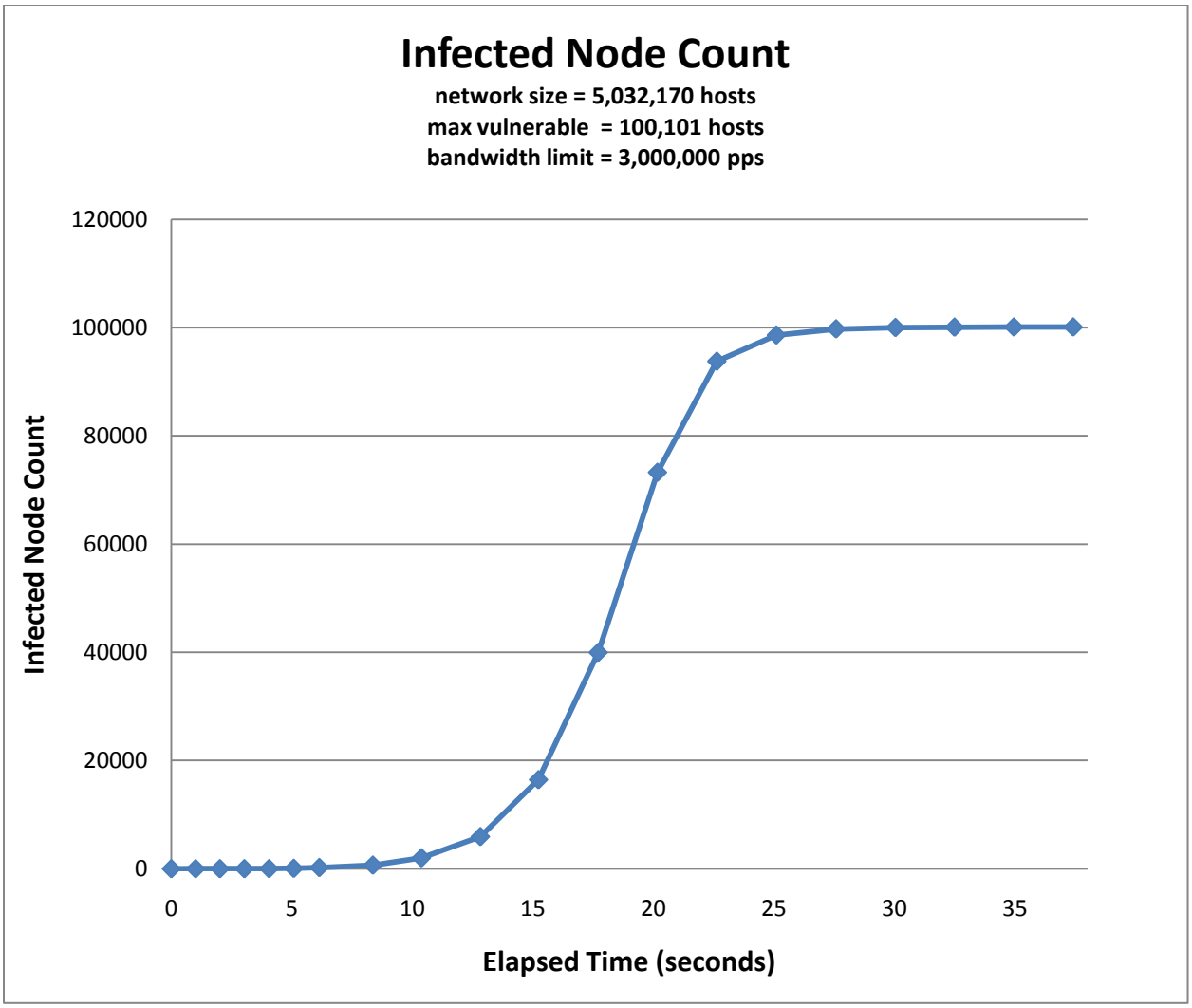

Figure 5.2-7 Infected Node Count – 5M Hosts, 100K Vulnerable, 3M Bandwidth

The third simulation's results are shown in Figures 5.2-6 and 5.2-7. For this simulation the network size is increased to a little over 5 million potential target hosts, and the bandwidth limit is kept at 3 million packets per second. With a network size of little over 5,000,000 hosts and only 100,000 vulnerable hosts, the infection probability is only 2%. Compared to simulation 2, simulation 3 takes nearly twice as much time for the worm traffic to reach maximum

bandwidth. Simulation 3 takes more than double the time as simulation 2 to reach maximum infection rate.

In all three simulations the worm spread is extremely fast – as evidenced by the very short total elapsed time in each simulation. There are a couple of reasons for this. Number one, the simulated target address range is extremely small compared to the entire Internet address range used by the real SQL Slammer worm. Number two, varying path delays, bottlenecks, varying network bandwidths, and other network limitations are not taken into account in the simulation.

5.3 **Performance**

The simulations were run on a machine equipped with a 3.73 GHz dual-core processor and 2 GB of physical memory. Running 1,000 concurrent threads on this machine consumed a max of 400 MB of physical memory and fluctuated between 30-78% processor utilization. With these performance metrics, the simulation slows the computer a little during peak processor usage; otherwise the computer had no problems performing other typical tasks while the simulation was running.

The simulation timescale factor turned out to be closer to 140 to 1, instead of the theoretical 100 to 1 timescale factor mentioned in the implementation section. For example, simulation 1 took 46 minutes to simulate 20 seconds. The 100 to 1 timescale factor suggested earlier did not account for the time it would take for the SCU to examine all of the packets on the IPQ during the dequeuing phase of each iteration.

6 Future Work

The Improved Worm Simulator can be used as a foundation for further work in resource limited packet level worm simulation. Several improvements are suggested here. The Improved Worm Simulator can technically simulate large scale networks with millions of potential targets, but it is still not practical to simulate more than a couple hundred thousand *active* worm nodes. An obvious improvement would be to distribute the simulation workload across many physical hosts. However, to keep in line with the affordability and accessibility goals mentioned earlier, it is suggested that the number of physical hosts be limited to 3 to 5 physical hosts.

The Improved Worm Simulator is flexible and scalable, but it also simplifies many aspects of a real network. It does not account for things such as network topology, path delays, and varying bandwidths across the network. These network characteristics can be stored in a local database, where it can be quickly queried and used to simulate a more realistic network. Another thing that the Improved Worm Simulator does not account for is competing network traffic. For this, a separate physical host can be dedicated to send packets representing competing network traffic.

Another improvement that can be made is to make the simulation more portable. Currently, the Improved Worm Simulator is implemented for the Linux platform only. Although the simulator is written entirely in C++, the simulator has a high dependency on the use of the libipq library, which is implemented for the Linux kernel only. Because of this high dependency on the libipq library, making the Improved Worm Simulator portable might be a substantial undertaking. However, it would definitely be worth considering.

7 Conclusion

Attempting to implement a worm simulator that can be run on a single machine proved to be a challenge given the complexities of multi-thread programming and the limited memory and processor resources. However, the implementation of the Improved Worm Simulator was a success. The simulation results showed that the Improved Worm Simulator is able to scale to at least 100,000 active worm nodes, and that it correctly models the behavior of the SQL Slammer worm.

The Improved Worm Simulator is highly configurable and scalable. The scalability, however, is really dependent upon the patience and time constraints of the user. If the user can tolerate hours or even days of simulation time, then the Improved Worm Simulator can scale to hundreds of thousands of infected nodes – perhaps even a million. If the user cannot afford long simulation times, then the Improved Worm Simulator can still manage a reasonable scale of at least 100,000 infected nodes.

In its current form, the Improved Worm Simulator can easily be used for instructional purposes. Students can experiment with various configurations and observe the resulting changes in worm behavior. Students can also modify the code to model other types of worms. The Improved Worm Simulator can be used as a research tool as well. Researchers can use the current framework to implement and test various defensive strategies against various worm spread models.

References

[1] Jyotsna Krishnaswamy, "Wormulator : Simulator for Rapidly Spreading Malware," Computer Science Graduate Writing Project, San Jose State University, 2009

[2] Mark Stamp, "Information Security Principles and Practice," Hoboken, NJ: Wiley, 2006.

[3] John Aycock, Heather Crawford, Rennie deGraaf, "Spamulator: The Internet on a Laptop," in Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education, Madrid, Spain, 2008, pp. 142-147.

[4] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, Nicholas Weaver, "Inside the Slammer Worm," IEEE Security and Privacy, v. 1, n. 4, pp. 33 – 39, 2003.

[5] John Markoff, "Worm Infects Millions of Computers Worldwide," Feb 2009. [Online]. URL: http://www.nytimes.com/2009/01/23/technology/internet/23worm.html. (Accessed: Jan 26, 2010)

[6] Stuart Staniford, David Moore, Vern Paxson, Nicholas Weaver, "The Top Speed of Flash Worms," in Proceedings of the 2004 ACM workshop on Rapid Malcode, Washington DC, USA, n.2, 2004, pp. 33 - 42.

[7] Article featured in Malware and Hardware Security magazine, 02 November 2009. [Online]. URL: http://www.infosecurity-magazine.com/view/4934/information-securitythreats-in-h1-2009-malware-and-rogue-security-software/. (Accessed: Jan 26, 2010)

[8] Article featured in Network World magazine, 30 October 2009. [Online]. URL: http://www.networkworld.com/news/2009/103009-after-one-year-conficker-infects.html. (Accessed: Feb 13, 2010)

[9] CERT advisory CA-2003-04, "MS-SQL server worm," [Online]. URL: http://www.cert.org/advisories/CA-2003-04.html. (Accessed: Jan. 26 2010)

[10] Robert Beverly, Arthur Berger, Young Hyun, K. Claffy, "Understanding the efficacy of deployed internet source address validation filtering," in IMC '09: Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference," Chicago, Illinois, USA, November 2009, Pages: 356-369.

[11] Miad Faezipour, Mehrdad Nourani, Rina Panigrahy, "A hardware platform for efficient worm outbreak detection," Transactions on Design Automation of Electronic Systems (TODAES), Volume 14, Issue 4, August 2009.

[12] Wikipedia The Free Encyclopedia, "Timeline of computer viruses and worms." Wikipedia, [Online]. URL: http://en.wikipedia.org/wiki/Timeline_of_notable_computer_viruses_and_worms . (Accessed: Feb 13, 2010)

[13] Cynthia Wong , Stan Bielski , Jonathan M. McCune , Chenxi Wang, "A study of mass-mailing worms," In Proceedings of the 2004 ACM workshop on Rapid malcode, October 29-29, 2004

[14] Yi Tang, Xiangning Dong, "Anting: An Adaptive Scanning Method for Computer Worms," In Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence (WI '06). IEEE Computer Society, Washington, DC, USA, Pages: 926-932

[15] Microsoft Security Intelligence Report Volume 9 (January through June 2010), [Online]. URL: http://www.microsoft.com/security/sir/keyfindings/default.aspx . (Accessed: Nov 20, 2010)

[16] Duc T. Ha, Hung Q. Ngo., "On the trade-off between speed and resiliency of flashworms and similar malcodes," In Proceedings of the 2007 ACM workshop on Recurring malcode (WORM '07). ACM, New York, NY, USA, Pages: 23-30

[17] Songjie Wei, Jelena Mirkovic, Martin Swany, "Distributed Worm Simulation with a Realistic Internet Model," In Proceedings of the 19th Workshop on Principles of Advanced and Distributed Simulation (PADS '05). IEEE Computer Society, Washington, DC, USA, Pages: 71-79

[18] Dancho Danchev, "Conficker's estimated economic cost? $9.1 billion," 23 April 2009. [Online]. URL: http://www.zdnet.com/blog/security/confickers-estimated-economic-cost-91-billion/3207. (Accessed: October 20, 2010)

[19] Paul Boutin, "Slammed," July 2003. [Online]. URL: http://www.wired.com/wired/archive/11.07/slammer.html. (Accessed: January 26, 2010)

[20] Robert Lemos, "Counting the cost of Slammer," [Online]. URL: http://news.cnet.com/Counting-the-cost-of-Slammer/2100-1001_3-982955.html. (Accessed: January 26, 2010)

[21] Cisco Systems, "Portable Product Sheets – Routing Performance," [Online]. URL: http://www.cisco.com/web/partners/downloads/765/tools/quickreference/routerperformance.pdf. (Accessed: October 26, 2010)

[22] John Aycock, "Computer Viruses and Malware," Advances in Information Security, volume 22, Springer-Verlag, 2006.