2009

Wormulator: Simulator for Rapidly Spreading Malware

Jyotsna Krishnaswamy
San Jose State University

WORMULATOR :

SIMULATOR FOR

RAPIDLY SPREADING MALWARE

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

by

Jyotsna Krishnaswamy

December, 2009

SAN JOSE STATE UNIVERSITY

The Undersigned Project Committee Approves the Project Titled

WORMULATOR: A SIMULATOR FOR RAPIDLY SPREADING MALWARE

by

Jyotsna Krishnaswamy

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

_____

Dr. Mark Stamp, Department of Computer Science Date

_____

Dr. Melody Moh, Department of Computer Science Date

_____

Dr. Teng Moh, Department of Computer Science Date

Abstract

This project addresses the need for an application level simulator to simulate Internet-wide phenomenon such as flash worms, botnets, Distributed Denial-of-Service attacks, etc. There are many network simulators intended for parallel and distributed simulation, but most are designed to simulate low level communication protocols such as TCP/IP. The desire to simulate rapidly spreading malware for research and teaching purposes lead us to explore the Spamulator, which was designed to simulate spam email on an Internet-wide scale. The Spamulator was developed by a team at the University of Calgary. It is a lightweight, application level simulator, which implements limited set of features of the Internet. In this project, the Spamulator is enhanced with the User Datagram Protocol (UDP) to simulate UDP worms. The modified version of the Spamulator is called the Wormulator. Wormulator tracks instantaneous network traffic, identifies and signals congestion throughout the network. The Wormulator is further enhanced with the use of POSIX threads instead of forking processes to create a distributed network of simulated servers. The resulting tool is called the "Enhanced Wormulator". Finally, a random scanning UDP worm with behavior similar to the well known SQL Slammer worm is modeled to validate the results of our simulation.

Results and data gathered from the simulation exhibit a qualitative resemblance to the real-world SQL Slammer worm. "Enhanced Wormulator", which uses POSIX thread instead of forking a process, had a catalytic effect on the scalability factor of the simulation. The simulation was run on a network of 30,000 server nodes. Hence, we conclude that rapidly spreading malware can be effectively simulated using the Wormulator.

Acknowledgement

I take this opportunity to thank Dr. Mark Stamp, my project adviser, for his guidance and support throughout my Master's degree and my project. I also thank my professors, Dr. Melody Moh and Dr. Teng Moh for serving as my committee. I am grateful to Dr. Kenneth Louden and Dr. Horstmann for their support. I would like to mention and thank Dr. John Aycock for sharing his work, the Spamulator, with us. Thanks to the Computer Science Department and the Graduate School of San Jose State University for their unremitting patience and support.

I would like to thank my loving husband Vijay for believing in me and encouraging me in all my endeavors. I thank my mom, dad and brother for providing me with the courage to realize my potential. I thank my Uncle Ravi and Aunt Claire for their love and support. Last but not the least, this project would not have been a success if not for my friends, who have always been around for thought provoking discussions. I thank Aditya, Vinay, and Preeti for their timely help and support in my work.

Table of Contents

Chapter 1

Introduction

Computer worms exhibiting a swarm-like behavior continue to beleaguer the Internet. A computer worm is a self-propagating malicious software program, which spreads on a network by exploiting some vulnerability on the target hosts. The worm may not alter target hosts, but it will likely disrupt network traffic. According to [2], a well-designed worm could infect the entire Internet in 15 seconds.

SQL Slammer worm (sometimes called Sapphire) was the fastest computer worm in history. The worm, released on January 2003, exploited stack buffer overflows in MS SQL server and MSDE engines. It is said to have infected 90 percent of the vulnerable hosts in less than 10 minutes. The most novel feature of the worm is its speed, which is attributed to its size – it fits in a single UDP packet. SQL Slammer worm uses random scanning to spread itself. The worm randomly generates an Internet address to find the next vulnerable target. Random scanning worms are only a subset of known worm spread algorithms [5].

Unlike random scanning worms, Flash worms adopt a hit-list scanning strategy. The author of the worm collects a list of potentially vulnerable machines before the worm is released. According to [1], the list is used to compute an efficient spread tree, which is encoded in to the worm. When the flash worm is released on to an initial machine, it scans down the collected list of vulnerable machines. When it infects a machine it divides the hit list in half, communicating half of the list to the recipient worm, and keeping the other half of the list [3]. Even though Flash worms are yet to be seen in reality, they are of importance for two reasons [1]:

- Flash worms are fastest possible worms.

- Given the off-line nature of the computation of the worm spread map, Flash worms are of interest for studying and exploring containment defenses.

Topological scanning uses information contained on the victim machine to harvest a new target. Email worms have used this tactic to harvest potential targets. A worm attacking a web server could use the URLs stored on the server to find potential targets. Morris worm used this technique. Code Red II employed local subnet scanning. Instead of selecting machines at random, the worm scanned for targets on local addresses, those which were identical in the upper address range [3].

Worms such as Conficker (January 2009) not only spiraled around the Internet at lightning speed, but also harnessed infected computers into unified systems, called botnets [7]. The term botnet is used to define a network of infected hosts, called bots, which are under the control of a human called the botmaster. Botnets recruit vulnerable machines by remotely exploiting software vulnerabilities by the use of worms, and social engineering, etc [4]. Botnets are largely used for criminal activities such as extortion, email spamming, identity theft, and software piracy [4].

The rise of malware on the Internet mandates the need to study and research rapidly spreading malware on an isolated test network. A framework must be laid to simulate and analyze the impact of malware on the Internet.

1.1 Purpose of this Project

The main objective of this project is to simulate rapidly spreading malware on the Internet. The requirements for the simulation are:

- A light weight application level network simulator.
- Simulate a limited set of Internet features such as network bandwidth, and the effects of congestion on the network.

- Model a real world example of rapidly spreading malware. We chose to simulate the SQL Slammer worm as there is abundant information and data available about the worm.
- Model a scalable network of nodes.
- Validate the results of our simulation with the actual data and behavior of the SQL Slammer worm.

We hope that our simulator can be used to research and teach Internet-wide phenomenon such as Distributed Denial-of-Service attacks, Botnets, XSS exploits, and develop and test innovative defense mechanisms.

1.2 Order of the Project

Chapter 2 discusses relevant work pertaining to worm simulation. Significant projects and their weakness are described. The justification for choosing Spamulator, a simulator to simulate email spam, as the basis of our worm simulation is discussed.

Chapter 3 discusses the details of the Spamulator and the necessary modifications to simulate rapidly spreading malware.

Chapter 4 describes the modified version of the Spamulator called the Wormulator. This chapter also discusses other implementation details external to the Wormulator but essential for the simulation, such as simulating a scalable distributed network of nodes, modeling network bandwidth and effects of congestion on the network, and modeling a random scanning UDP worm.

Chapter 5 describes the "Enhanced Wormulator", which is an improvement over the "Wormulator" in Chapter 4. The improvement is mainly in the scalability factor of the number of server nodes and simplification of the complexities built into the Spamulator/Wormulator.

Chapter 6 deals with the simulation itself. This chapter discusses the experimental set up, followed by a preview of the simulation and data collection, and analysis and validation of the results.

Chapter 7 elaborates on our approach to the problem of malware, limitations of our simulation, and the possible future work.

Chapter 8 is the concluding chapter which reiterates our goals, achievements and contribution to the field of computer network security and study of malware.

Appendix A provides background information on IP Tables and library libIPQ.

Appendix B discusses various Inter Process Communication Mechanisms used in the simulation.

Appendix C describes the POSIX threads used in the simulation.

Chapter 2

Related Work

The work for this project can be categorized in three separate parts: modeling the SQL Slammer worm, using a loop back network simulator, and modeling a limited set of features of the Internet. The SQL Slammer exemplifies rapidly spreading malware, and the Wormulator is proof of concept of simulating rapidly spreading malware. The following are associated work, which serve as an insight to this project work.

2.1 Traditional Network Simulators

There are different network simulators for different types of networks. SENSE is a wireless network simulator for sensor networks. It frees simulation models from interdependence usually found in an object-oriented architecture, and promotes reusability, scalability and extensibility [8]. NeuroWeb is an Internet-based framework for the simulation of neural networks. It aims for using the Internet as a transparent environment to allow users the exchange of information (neural network objects, neural network paradigms) and the exploit of available computing resources for neural network specific tasks [9].

Network Simulator (NS-2) is a popular network simulation tool for TCP/IP protocols and algorithms. It is an object-oriented simulator that allows users to create realistic network topology, network components such as routers, hosts, and monitor packet queue at the nodes, and simulate wireless mobile nodes [10]. Although NS2 is widely adopted, there are some limitations to it. First, it does not have the functions related with real IP addresses. Second, its agent mechanism makes it inconvenient to configure a simulation task, such as the TCP connection behaviors [11].

OPNET, developed by OPNET technologies is a popular commercial tool used for TCP/IP network simulation. It is an object-oriented and menu driven simulator with a user friendly Graphical User Interface (GUI) [12]. OPNET is expensive making it infeasible for use in the academic world. The SENSE, Neuro Web, NS-2, and OPNET simulators are sequential programs that run on a single machine. This is not suitable to simulate malware, which propagates on a distributed network such as the Internet.

2.2 Parallel/Distributed Network Simulators

To overcome the short comings of a sequential simulator, a team of researchers from Georgia Institute of Technology developed a parallel library called libSynk, and used it to build the first parallel discrete event simulator – GTNetS. The library libSynk was used to develop a parallel version of NS2 called PDNS. GTNetS supports a large variety of TCP and UDP based applications, TCP/IP/MAC layer protocols, mobile nodes, routing algorithms and distributed simulation of a topology on network of workstations, a shared-memory symmetric multiprocessing system, or a combination of both [14].

GloMoSim is a sequential/parallel library developed at University of California Los Angeles, for simulation of large-scale Wireless Networks. Each module of the library simulates a specific wireless communication protocol in the protocol stack [15]. According to [11], the above simulators were originally designed to simulate and test communication protocols such as MPLS and TCP/IP. They are not convenient for application layer simulation as they focus on lower levels of the TCP/IP stack.

2.3 High Level, Application Level, Parallel/Distributed Worm Simulators

The first attempt to design a simulator for applications is the SimGrid, a simulation-based framework for evaluating cluster, grid and P2P algorithms, and heuristics. SimGrid provides several programming environments to develop a real distributed application, study the behavior of a MPI application, and study theoretical problems and compare several heuristics [16]. However, this is not a universal simulator for other distributed applications.

A more sophisticated and dedicated worm spread simulator called PAWS, developed by a team from University of Delaware, runs on multiple PCs and models a realistic internet topology, background traffic, and link bandwidth to capture effects of congestion. However, this is a packet level simulation and requires a test bed such as Emulab to run the simulation [13]. Emulab is an experimental network platform available to remote users [17]. According to [6], Emulab faces scalability problems along with the risk of allowing a secure network have an external network connection. An alternative to Emulab is virtualization provided by VMware, which hosts multiple virtual machines on a single physical machine. This mechanism is resource-intensive and does not scale well [18].

2.4 Spamulator

John Aycock, Heather Crawford, and Rennie deGraaf, Department of Computer Science, University of Calgary developed an Internet simulator – the Spamulator to teach a course on spam and spy ware [6]. The Spamulator is a lightweight network simulator running on a single machine. According to [6], The Internet is a complicated thing; we do not need to simulate the entire Internet, just those parts of it which are necessary for sending spam.

Thus, the Spamulator implements a limited set of features of the Internet such as Network Routing Daemon for Transport Control Protocol and Domain Name Server for simulated servers. The Spamulator is designed to simulate an Internet with millions of domains, work alongside normal Internet applications, function under extremely heavy use, and is extensible for future projects and research [6]. The Spamulator requires little or no physical hardware requirements as it can be run on a single machine or a laptop [6].

From the above discussions, we decided that the Spamulator is the ideal choice to simulate rapidly spreading malware for research and teaching purposes. An initial study of the tool revealed that it supports only TCP applications such as Telnet. Hence, it we decided to extend the Spamulator to provide UDP support. Additionally, the following features critical for the simulation of a UDP worms were added to the simulator: 1) track network traffic, and 2) simulate the effects of congestion in the network. The resulting simulator is called the Wormulator.

Chapter 3

Spamulator

This section begins with a level high architecture of the Spamulator, followed by implementation details, and ends with the necessary modifications of the tool for the purpose of simulating rapidly spreading malware.

3.1 Architecture

The Spamulator is a loop back network simulator, whose components run on a single computer machine or a laptop. The architecture of the Spamulator is illustrated in Fig 3-1.1.

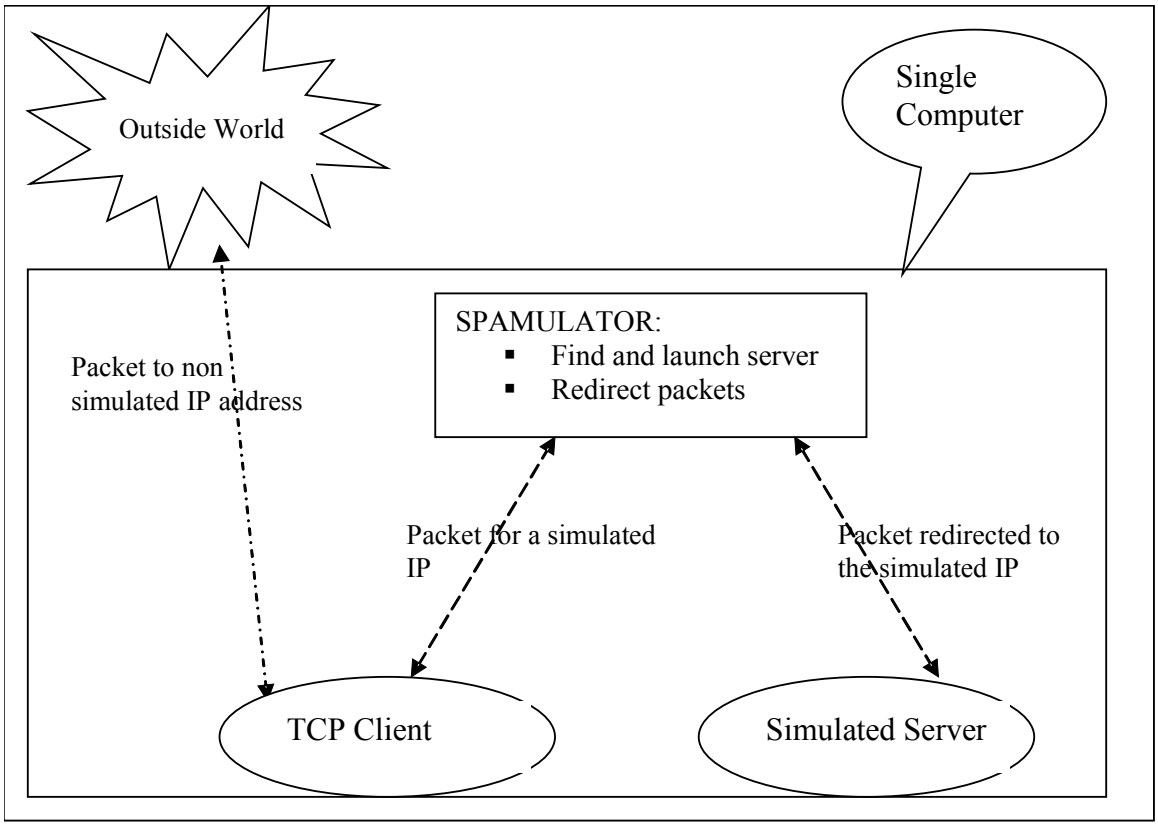

Figure 3-1.1 Architecture of Spamulator

Network packets that originate from a client program destined to a simulated server are redirected to a local queue.

The packets on the queue are read by core of the Spamulator, the Loop Back Network Simulator (LNS), which reroutes the packet to the local simulated server [6]. Return traffic from the simulated server to the client program is handled in a similar manner. Network traffic not destined to a simulated IP address is untouched by the Spamulator.

Apart from rerouting packets, the Spamulator finds and launches simulated servers, keeps track of open connections, forwards packets between a server and a client, obtains domain name information for the simulated servers from a local DNS, and performs clean up when a server ceases to exist; thereby, acting as the backbone of the simulation. The Spamulator makes use of the "Network Interface Card" (NIC), to send and receive packets. Therefore, the NIC must be active while executing the tool.

The underlying architectural design principle is simplicity. This is apparent in what has been excluded from the Spamulator – for instance, no attempt was made to simulate network topology, latency, or failures [6].

3.2 Implementation Details

Spamulator has been implemented on two platforms: Linux and Mac OS [6]. Our simulation is built on Linux platform and uses the Linux version of the Spamulator. Spamulator makes use of the Linux packet-filtering rules: 1) MANGLE IP Table rules ensure that packets destined for the simulated servers are queued on to a local queue instead of being sent out [19]. 2) Packets on the queue are accessed by the Spamulator via the LibIPQ library [20]. See Appendix A for more details on IP Tables and the library LibIPQ.

The core functionality of the Spamulator, packet handling, is managed by the LNS module. The pseudo code for LNS is given in Figure 3-2.1. When LNS receives a packet, a new connection is detected by seeing if P is a SYN packet. The IP address 127.0.0.1 is the loop back address of the local machine and the IP address 127.0.0.2 is used as a sentinel value to detect return traffic [6]. The simulated server is launched by forking a child process. A pipe is created between the Spamulator and the child process to communicate the server's port number. When the server writes back, a SIGIO signal is raised and handled by the LNS module.

```
Create Stream socket

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

if P is a new connection:
      find server to handle connection
      start server process
      wait for port number P'd from simulated server
      store (As, Ps, Ad, Pd, P'd) in table T
      rewrite P into P:
         ▪ change As to 127.0.0.2
         ▪ change Ad to 127.0.0.1
         ▪ change Pd to P'd
else:
      if As = 127.0.0.1 and Ad = 127.0.0.2:
         ▪ this is return traffic from the server
      find entry (?,?,?,?,Ps) in T
      rewrite P into P':
         ▪ change Ps to Pd found in T
         ▪ change As to Ad found in T
         ▪ change Ad to As found in T
else:
      find entry (As, Ps, Ad, Pd,?) in T
      rewrite P into P':
         ▪ change As to 127.0.0.2
         ▪ change Ad to 127.0.0.1
         ▪ change Pd to P'd found in T

send P'
```

Figure 3-2.1 Anatomy of the LNS - Spamulator [6]

The server mentioned in the code in Figure 3-2.1 is a stand alone C++ executable or a script present under '/var/lns/useservers/' directory. When LNS detects a new connection, it follows the algorithm in Figure 3-2.2 to locate a server.

```
▪ A is the destination IP address as a string, e.g., "10.0.0.1"
▪ P is the destination port as a string, e.g., "42"
▪ Du = /var/lns/useservers

if executable file Du/A:P exists: return Du/A:P

if executable file Du/A exists: return Du/A

do a DNS reverse lookup
   ▪ destination IP address queried
   ▪ TXT record requested, not PTR record
return SERVER_NOT_FOUND
```

Figure 3-2.2 Anatomy of locating a server [6]

The Spamulator communicates with the servers through various Inter Process Communication techniques: 1) each server is executed as a child process created by fork and exec system calls. A pipe is opened between the server and the Spamulator, the read end of the pipe is set to the Spamulator and the write end is set to the server. 2) Packets intercepted by the Spamulator are forwarded to the simulated server by network sockets. 3) Signals are used for asynchronous events between the Spamulator and its child processes- the simulated servers. See Appendix B for more details on IPC used in the simulation.

3.3 Proposed Modifications

One of the goals of this project is to simulate the SQL Slammer worm, which is a UDP worm. The Spamulator implements only Transmission Control Protocol. The first modification is to implement User Datagram Protocol.

The SQL Slammer worm contributed to an explosive growth in network traffic across the Internet resulting congestion in the network. To simulate the same network congestion, the second modification to the Spamulator is to measure the instantaneous network traffic and model congestion in the network.

A TCP application engages in a one-to-one dedicated connection between a client and a server, which ceases to exist when a FIN packet is received. A new connection needs to be established for subsequent communication. Our simulation differs from TCP applications in many aspects: 1) A UDP application does not establish connection, 2) A UDP engages in many-to-one connection where many clients can send a packet to one server, and 3) A UDP employs one-way communication, from an infected machine to a potential target. Given these differences, the third modification to the Spamulator is to store a table of open connections based on the destination IP address and port number only unlike TCP connections, which store the source and the destination details to identify a connection uniquely.

The modified version of the Spamulator is called the Wormulator, which is discussed in the following Chapter 4.

Chapter 4

Wormulator

From Chapter 3 it is evident that the Spamulator cannot be used in its original form to simulate rapidly spreading malware. This section begins with a high level architecture of the Wormulator, followed by implementation details, and ends with the discussion of requirements extraneous to the Wormulator, but essential for the simulation of the SQL Slammer worm.

4.1 Architecture

The Wormulator is an extension of the Spamulator with the addition of User Datagram Protocol (UDP), measurement of instantaneous network traffic, and simulate effects of congestion in the network. The architecture of the Wormulator is illustrated in Fig 4-1.1.

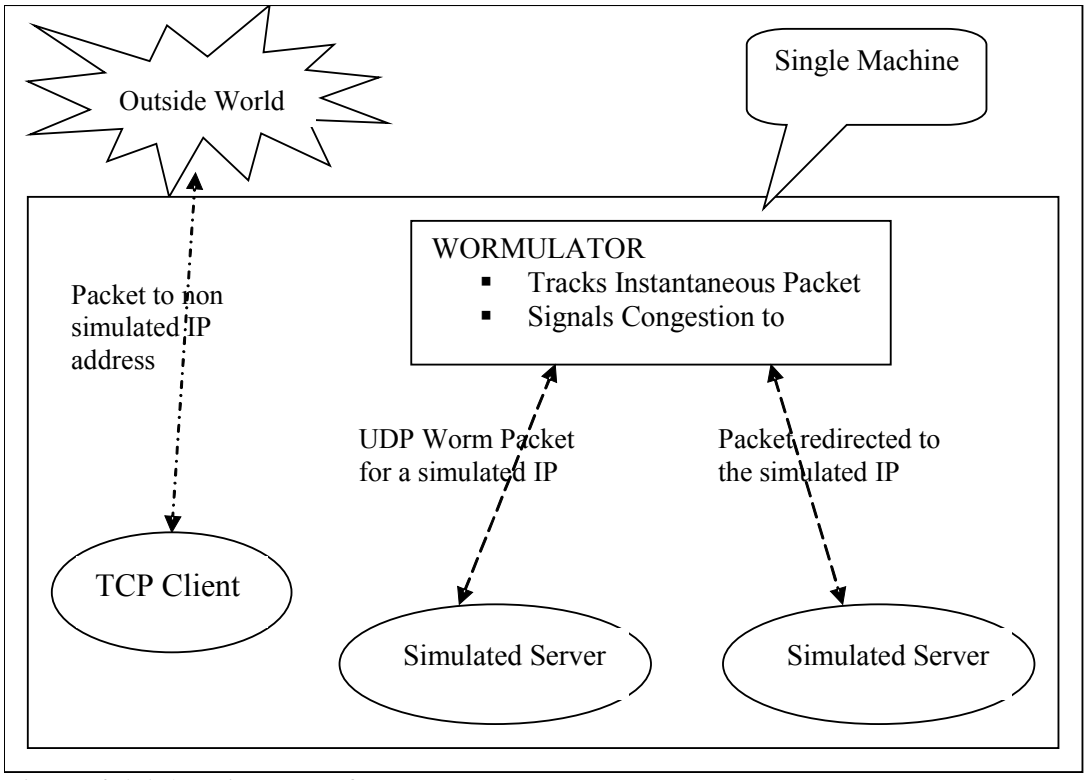

Figure 4-1.1 Architecture of Wormulator

The Wormulator is a modified version of the Spamulator providing User Datagram Protocol to simulate UDP applications such as the UPD worm simulation. Since the Spamulator supports connection-oriented TCP, we decided to make the Wormulator provide connectionless UDP support. Study of the Spamulator revealed that the most significant modifications were to be made in the core functionality of the Spamulator, the LNS module. All the other modules of the Spamulator are included in the Wormulator and used as needed. Wormulator is implemented by making minimal changes to the original design of the Spamulator.

Apart from providing UPD support, the Wormulator measures the instantaneous network traffic, i.e. the number of packets in the network at a given instant of time. The Wormulator signals congestion to all the simulated servers, when the network traffic generated by the worm increases beyond a pre-defined limit. This is essential to capture the consequence of the UDP worm on the network. See Appendix B for more details on the inter-process communication mechanisms employed by the Wormulator.

4.2 Implementation Details

Wormulator is implemented on the Linux platform and uses the Linux version of the Spamulator. The core functionality of the Spamulator, packet handling, managed by the LNS module is modified to implement UDP in the Wormulator. The pseudo code for the modified LNS with UDP is given in Figure 4-2.1. The IP address 127.0.0.1 is the loop back address of the local machine. Since there is no return traffic in the simulation of the UDP worm, there is no need for a sentinel value to detect return traffic. The Wormulator creates a datagram socket to read UDP packets. An instant of time is measured by invoking ‘time (NULL)’ system call of Linux. If the current time differs from the previous time, a new instance is determined.

User defined signals are used to simulate the effect of congestion in the network. SIGUSR1 signals a congestion free network and SIGUSR2 indicates congestion in the network. The servers that receive these signals control the rate of worm scan traffic accordingly. The total network bandwidth is configurable.

```
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:
      find server to handle connection
      start server process
      wait for port number P'd from simulated server
      store (Ad, Pd, P'd) in table T
      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
else:
      send signal SIGUSR1 to all servers

send P'
```

Figure 4-2.1 Anatomy of the modified LNS – Wormulator

When LNS receives a UDP packet, a new connection is detected by looking for an open connection that matches the destination server IP address and port number. A significant deviation from the Spamulator is in the particulars stored in connection table T in Figure 4-2.1. An existing connection is uniquely identified by the destination address only and not by a combination of source and destination address as in the Spamulator.

4.3 Requirements Extraneous to the Wormulator

To successfully simulate rapidly spreading malware similar to the real world malware, we must implement additional features, which are external to the Wormulator but crucial for an accurate simulation.

First and foremost, a distributed network of nodes is created to replicate the real world SQL servers, which were targeted by the SQL Slammer worm. The worm on an infected server generates random IP addresses and sends itself to these addresses. Hence, each server node is also capable of sending worm packets to other servers. The mode of communication between these nodes is via message passing in an asynchronous mode, which is similar to the Internet. A server is a C++ executable named after a unique IP address and port number. The pseudo code of a simulated server is given in Figure 4-3.1. The empty packet serves to bring the server alive before initiating the worm spread. When a packet is sent for the first time to a server, the Wormulator captures the packet and launches the server. The signals SIGUSR1 and SIGUSR2 are used to control the rate of worm scan traffic. See figure 4-3.2 for more details on how this is done. See The worm packet, if received, is handled by an independent thread. See Appendix C for more details on pthreads.

```
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 4-3.1 Anatomy of a simulated server

```
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 4-3.2 Anatomy of a UDP worm

While the SQL Slammer worm had no malicious payload, it caused considerable harm by overloading networks and disabling database servers. Many sites lost connectivity as local copies of the worm saturated their access bandwidths [5].

Once a machine is compromised, the worm tries to propagate itself. A random scanning UDP worm with behavior similar to the SQL worm is modeled for our simulation. The pseudo code of random scanning UDP worm is given in Figure 4-3.2. In our worm model, the worm sends itself by sending a single UDP packet containing the absolute path of the worm executable. Random IP addresses are restricted to the simulated servers only and are generated using the 'boost::random' number generator. The variable 'sleepRange' plays an important role in controlling the worm scan traffic. The sleep command prevents dominance of a single instance of the worm. The variable 'sleepRange' is modified dynamically. When congestion is detected in the network, as indicated by SIGUSR1 in Figure 4-3.1, the sleepRange will increase at every new instant of time. This will result in a gradual decline in the network traffic. When the network is free of congestion, as indicated by SIGUSR2 in Figure 4-3.1, the sleepRange will decrease at every instant of time, resulting in an exponential growth in network traffic. The variable 'sleepRange,' is modified by the formula:

```
VARIANCE * GAUSSIAN_GENERATOR() + FACTOR
```

VARIANCE and boost::GAUSSIAN_GENERATOR() are used to introduce randomness in the pattern, and FACTOR is the factor by which the sleepRange is increased or decreased. The initial value of sleepRange of 100 micro seconds is effective for a test bed of 2,500 nodes. Increase the sleepRange to 200 micro seconds for a test bed greater than 2,500. The simulation could be successfully scaled to a test bed of 3,500 server nodes.

4.4 Limitations of the Wormulator

The main objective of this project is to simulate the SQL Slammer worm on a scalable network of nodes, thousands of server nodes running on a single machine. This set up is necessary to accurately demonstrate the effects of the worm on the Internet.

A key feature that influences the performance and scalability of the Wormulator is the way the tool creates and manages its children - the simulated servers. Each server is spawned as a separate child process. A huge overhead is incurred in creating and maintaining a process. A significant improvement can be achieved by creating light weight, resource conservative POSIX threads as a replacement for processes. This would entail fundamental changes in the Wormulator. The primary motivation for using Pthreads is to realize potential program performance gains [30]. Multi threading can pose its own share of challenges and special caution must be exercised to synchronize communication between the threads, if needed. Chapter 5, discusses the challenges and the details of the “Enhanced Wormulator”, which implements threads instead of processes.

Chapter 5

Enhanced Wormulator

The Wormulator, which spawns the server nodes by forking a separate child process, does not scale beyond 3,500 server nodes. It was observed that the CPU context switch between these processes was memory and time intensive. This lead to the exploration of the possibility of using threads instead of processes to create the server nodes. This section describes the implementation details of the "Enhanced Wormulator" followed by a description of the challenges and solutions to creating several thousand threads by a single program – the "Enhanced Wormulator".

5.1 Implementation Details

Enhanced Wormulator is implemented on the Linux platform. The core functionality of the "Enhanced Wormulator", packet handling, managed by the LNS module is modified to create servers via the POSIX threads. The pseudo code for the modified LNS with UDP is given in Figure 5-1.1. A new thread is created when a packet for a simulated server is received for the first time. The thread will execute the server code described in figure 4-3.1.

The significant differences between the "Enhanced Wormulator" and the Wormulator/Spamulator are: 1) Light weight POSIX thread is created to spawn a server. 2) The newly created thread becomes a peer of the main thread; therefore, there is no hierarchy between threads and no concept of parent and child relationship. 3) The unique server port is known in advance and so the server does not write its port number to the LNS. 4) The server executable need not be created before hand under '/var/lns/userserver', as each server is executed as a thread. Consequently, there is no method to locate a server.

```
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 5-1.1 Anatomy of the LNS of Enhanced Wormulator

5.2 Challenges of Multi Threading

It is uncommon and not recommended to create several thousand threads from a single program. If a program contains tasks that need to be executed concurrently a static pool of threads is suggested.

However, this is unsuitable for our requirement of simulating an internet of nodes. So this project undertakes the daunting task of creating thousands of peer threads that continue to exist throughout the simulation.

The foremost challenge is to decide on an optimal stack size for each thread. Since, the main memory is limited and is shared by all the threads, the stack size for each thread it set to a minimum value so that a large number of threads can be created. It was observed that a minimum size of 80 -100 KB was necessary to run the simulation. By default, the stack size is larger than this value. So the operating system must be configured accordingly. See section 6.1 for more details. Furthermore, shared resources between threads must be synchronized. However, in our simulation no resource is shared among the threads.

Chapter 6

Simulation

This section begins with a description of the set up for the simulation using the "Enhanced Wormulator", followed by the experiment itself, and ends with an analysis and validation of the results of the experiment.

6.1 Experimental Set Up

The experimental set up consists of: 1) configure the Linux operating system, 2) install and configure the Wormulator, 3) configure rules for the network stack, and 4) create an isolated network of servers.

The default system configuration is modified as shown in Figure 6-1.1 to accommodate the creation of large number of threads, files and UDP packets generated during the simulation.

```
Increase number of threads:
     echo 100000000 > /proc/sys/kernel/threads-max

Append the following in '/etc/sysctl.conf' file:
   Increase the maximum default receive socket buffer size:
      ▪ net.core.rmem_max = 524280
      ▪ net.core.rmem_default = 524280

      Increase the maximum default send socket buffer size :
      ▪ net.core.wmem_max = 524280
      ▪ net.core.wmem_default = 524280

      Set the maximum number of open files:
      ▪ fs.file-max = 100000
```

```
#stack size for each thread
ulimit -s 80

#max open files
ulimit -n 40240
```

Figure 6-1.1 Configure features of the Linux Operating System

The Wormulator can be installed using the install document of the Spamulator. '/var/lns/useservers' directory is searched by the Wormulator to launch a server. Dependency packages are ‘libboost-dev' and 'iptables-dev'. The directory structure of the Spamulator is retained, except for the LNS module of the Spamulator is replaced by the modified LNS of the “Enhanced Wormulator”. A folder named ‘~/home/user/Files’ is created to store the log files created by the servers.

The Wormulator is configured by specifying a value for ‘`NETWORK_BANDWIDTH`’ in Figure 4-2.1. The value is configurable and is set to an initial value of 10000 in our simulation. Since the simulation uses the network protocol stack to send and receive packets, the Network Interface Card (NIC) of the machine needs to be active during the simulation.

The system network stack is configured to capture packets generated for simulated servers. Otherwise, the packets will be sent to the outside world. The output chain of the Mangle IP Table is modified by setting rules as described in Figure 6-1.2.1 and 6-1.2.2. See Appendix A for more details on IP Tables. The rules instruct the Mangle IP Table to redirect packets destined to simulated servers to a local queue, making the packets available to a user space application, in our simulation the Wormulator. Figure 6-1.2 captures packets destined for 30,000 servers.

```
iptables -t mangle -I OUTPUT -p udp -d 10.0.0.0/24 --destination-port 14000:14999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.1.0/24 --destination-port 14000:14999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.2.0/24 --destination-port 14000:14999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.3.0/24 --destination-port 15000:15999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.4.0/24 --destination-port 15000:15999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.5.0/24 --destination-port 15000:15999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.6.0/24 --destination-port 16000:16999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.7.0/24 --destination-port 16000:16999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.8.0/24 --destination-port 16000:16999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.9.0/24 --destination-port 17000:17999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.10.0/24 --destination-port 17000:17999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.11.0/24 --destination-port 17000:17999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.12.0/24 --destination-port 18000:18999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.13.0/24 --destination-port 18000:18999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.14.0/24 --destination-port 18000:18999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.15.0/24 --destination-port 19000:19999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.16.0/24 --destination-port 19000:19999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.17.0/24 --destination-port 19000:19999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.18.0/24 --destination-port 20000:20999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.19.0/24 --destination-port 20000:20999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.20.0/24 --destination-port 20000:20999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.21.0/24 --destination-port 21000:21999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.22.0/24 --destination-port 21000:21999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.23.0/24 --destination-port 21000:21999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.24.0/24 --destination-port 22000:22999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.25.0/24 --destination-port 22000:22999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.26.0/24 --destination-port 22000:22999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.27.0/24 --destination-port 23000:23999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.28.0/24 --destination-port 23000:23999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.29.0/24 --destination-port 23000:23999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.30.0/24 --destination-port 24000:24999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.31.0/24 --destination-port 24000:24999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.32.0/24 --destination-port 24000:24999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.33.0/24 --destination-port 25000:25999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.34.0/24 --destination-port 25000:25999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.35.0/24 --destination-port 25000:25999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.36.0/24 --destination-port 26000:26999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.37.0/24 --destination-port 26000:26999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.38.0/24 --destination-port 26000:26999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.39.0/24 --destination-port 27000:27999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.40.0/24 --destination-port 27000:27999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.41.0/24 --destination-port 27000:27999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.42.0/24 --destination-port 28000:28999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.43.0/24 --destination-port 28000:28999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.44.0/24 --destination-port 28000:28999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.45.0/24 --destination-port 29000:29999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.46.0/24 --destination-port 29000:29999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.47.0/24 --destination-port 29000:29999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.48.0/24 --destination-port 30000:30999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.49.0/24 --destination-port 30000:30999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.50.0/24 --destination-port 30000:30999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.51.0/24 --destination-port 31000:31999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.52.0/24 --destination-port 31000:31999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.53.0/24 --destination-port 31000:31999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.54.0/24 --destination-port 32000:32999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.55.0/24 --destination-port 32000:32999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.56.0/24 --destination-port 32000:32999 -j QUEUE
```

Figure 6-1.2.1 IP Table rules to capture packets sent to simulated servers

```
iptables -t mangle -I OUTPUT -p udp -d 10.0.57.0/24 --destination-port 33000:33999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.58.0/24 --destination-port 33000:33999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.59.0/24 --destination-port 33000:33999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.60.0/24 --destination-port 34000:34999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.61.0/24 --destination-port 34000:34999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.62.0/24 --destination-port 34000:34999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.63.0/24 --destination-port 35000:35999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.64.0/24 --destination-port 35000:35999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.65.0/24 --destination-port 35000:35999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.66.0/24 --destination-port 36000:36999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.67.0/24 --destination-port 36000:36999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.68.0/24 --destination-port 36000:36999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.69.0/24 --destination-port 37000:37999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.70.0/24 --destination-port 37000:37999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.71.0/24 --destination-port 37000:37999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.72.0/24 --destination-port 38000:38999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.73.0/24 --destination-port 38000:38999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.74.0/24 --destination-port 38000:38999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.75.0/24 --destination-port 39000:39999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.76.0/24 --destination-port 39000:39999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.77.0/24 --destination-port 39000:39999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.78.0/24 --destination-port 40000:40999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.79.0/24 --destination-port 40000:40999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.80.0/24 --destination-port 40000:40999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.81.0/24 --destination-port 41000:41999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.82.0/24 --destination-port 41000:41999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.83.0/24 --destination-port 41000:41999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.84.0/24 --destination-port 42000:42999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.85.0/24 --destination-port 42000:42999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.86.0/24 --destination-port 42000:42999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.87.0/24 --destination-port 43000:43999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.88.0/24 --destination-port 43000:43999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.89.0/24 --destination-port 43000:43999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.90.0/24 --destination-port 44000:44999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.91.0/24 --destination-port 44000:44999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.92.0/24 --destination-port 44000:44999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.93.0/24 --destination-port 45000:45999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.94.0/24 --destination-port 45000:45999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.95.0/24 --destination-port 45000:45999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.96.0/24 --destination-port 46000:46999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.97.0/24 --destination-port 46000:46999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.98.0/24 --destination-port 46000:46999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.99.0/24 --destination-port 47000:47999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.100.0/24 --destination-port 47000:47999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.101.0/24 --destination-port 47000:47999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.102.0/24 --destination-port 48000:48999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.103.0/24 --destination-port 48000:48999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.104.0/24 --destination-port 48000:48999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.105.0/24 --destination-port 49000:49999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.106.0/24 --destination-port 49000:49999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.107.0/24 --destination-port 49000:49999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.108.0/24 --destination-port 50000:50999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.109.0/24 --destination-port 50000:50999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.110.0/24 --destination-port 50000:50999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.111.0/24 --destination-port 51000:51999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.112.0/24 --destination-port 51000:51999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.113.0/24 --destination-port 51000:51999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.114.0/24 --destination-port 52000:52999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.115.0/24 --destination-port 52000:52999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.116.0/24 --destination-port 52000:52999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.117.0/24 --destination-port 53000:53999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.118.0/24 --destination-port 53000:53999 -j QUEUE
iptables -t mangle -I OUTPUT -p udp -d 10.0.119.0/24 --destination-port 53000:53999 -j QUEUE
```

Figure 6-1.2.2 IP Table rules to capture packets sent to simulated servers

An isolated network of servers is setup as follows: 1) the desired number of server executables is created under '/var/lns/useserver/' directory, 2) the Wormulator is executed in the background, and 3) the script in Figure 6-1.3 is interpreted. When a packet is sent to a non-existing simulated server, the Wormulator looks for the server in '/var/lns/useserver/' and if found launches the server and forwards the packet. The empty packet is ignored by the server. The sole purpose of the empty packet is to bring the server to life.

```
Create UDP socket

Loop though the number of simulated servers
   ▪ generate IP address of a simulated server, As
   ▪ calculate the corresponding port number, Ps
   ▪ send the empty UDP packet to As: Ps
```

Figure 6-1.3 Create network of servers

The sleepRange in the worm is modified depending on the number of server nodes. See figure 4-3.2 for more details.

6.2 Experiment

The simulated servers communicate to each other through the Wormulator. Figure 5-2.1 illustrates the experiment.

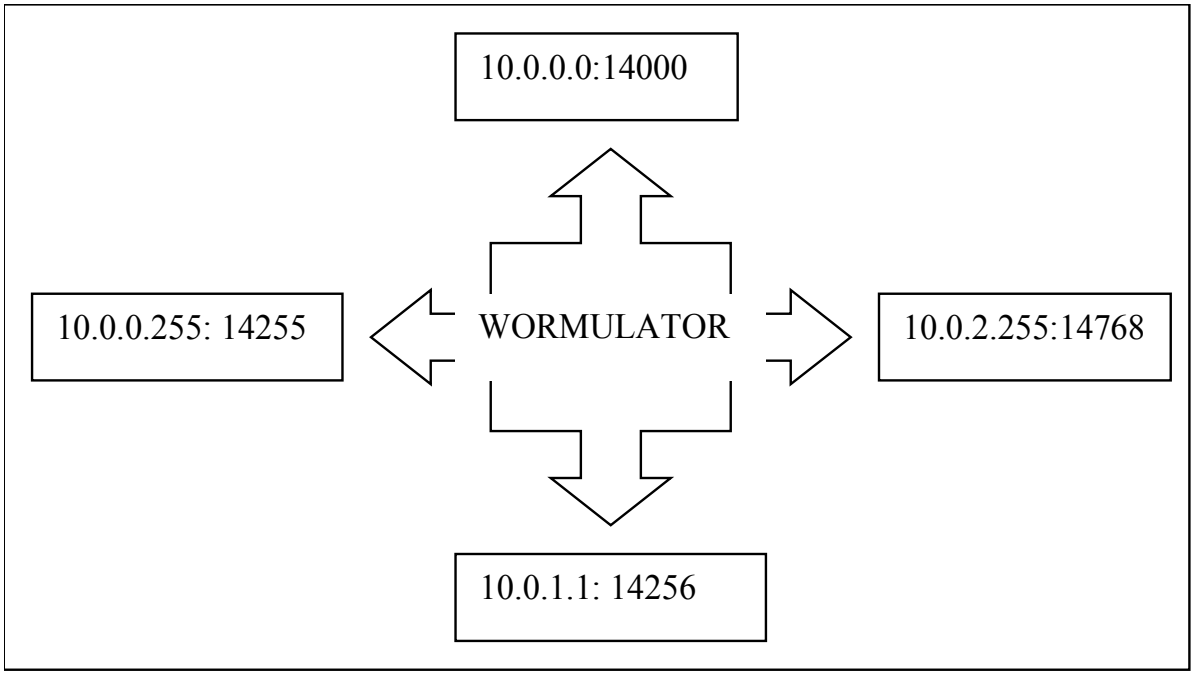

Figure 6-2.1 Create network of servers

The Wormulator executed in the verbose mode outputs various diagnostics. Diagnostics printed by the Wormulator, when the servers are launched is given in Figure 6-2.2. The diagnostics indicate successful launch of the simulated servers.

```
Oct 06 14:40:59 /usr/local/sbin/lns Info: Launching server /var/lns/uservers/10.0.0.0:14000
Oct 06 14:40:59 /usr/local/sbin/lns Info: Launching server /var/lns/uservers/10.0.0.1:14001
Oct 06 14:40:59 /usr/local/sbin/lns Info: Launching server /var/lns/uservers/10.0.0.2:14002
Oct 06 14:40:59 /usr/local/sbin/lns Info: Launching server /var/lns/uservers/10.0.0.3:14003
Oct 06 14:40:59 /usr/local/sbin/lns Info: Launching server /var/lns/uservers/10.0.0.4:14004
Oct 06 14:40:59 /usr/local/sbin/lns Info: Launching server /var/lns/uservers/10.0.0.5:14005
Oct 06 14:40:59 /usr/local/sbin/lns Info: Launching server /var/lns/uservers/10.0.0.6:14006
Oct 06 14:40:59 /usr/local/sbin/lns Info: Launching server /var/lns/uservers/10.0.0.7:14007
Oct 06 14:40:59 /usr/local/sbin/lns Info: Launching server /var/lns/uservers/10.0.0.8:14008
Oct 06 14:40:59 /usr/local/sbin/lns Info: Launching server /var/lns/uservers/10.0.0.9:14009
Oct 06 14:40:59 /usr/local/sbin/lns Info: Launching server /var/lns/uservers/10.0.0.10:14010
Oct 06 14:40:59 /usr/local/sbin/lns Info: Launching server /var/lns/uservers/10.0.0.11:14011
```

Figure 6-2.2 Snapshot of Spamulator launching servers [6]

Diagnostic in Figure 6-2.3 is printed when the worm packet is sent from one simulated server to another. The message indicates successful redirection of the packets to the destined simulated servers on the local host.

```
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.75:14075 to 127.0.0.1:1->127.0.0.1:14075
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.45:14045 to 127.0.0.1:1->127.0.0.1:14045
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.54:14054 to 127.0.0.1:1->127.0.0.1:14054
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.99:14099 to 127.0.0.1:1->127.0.0.1:14099
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.86:14086 to 127.0.0.1:1->127.0.0.1:14086
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.94:14094 to 127.0.0.1:1->127.0.0.1:14094
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.22:14022 to 127.0.0.1:1->127.0.0.1:14022
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.35:14035 to 127.0.0.1:1->127.0.0.1:14035
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.72:14072 to 127.0.0.1:1->127.0.0.1:14072
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.87:14087 to 127.0.0.1:1->127.0.0.1:14087
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.10:14010 to 127.0.0.1:1->127.0.0.1:14010
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.55:14055 to 127.0.0.1:1->127.0.0.1:14055
Rewriting address of UDP packet 192.168.1.65:55966->10.0.0.82:14082 to 127.0.0.1:1->127.0.0.1:14082
```

Figure 6-2.3: Snapshot of LNS of the Wormulator re-directing packets

The instantaneous network traffic measured and tracked by the Wormulator is printed on the console as depicted in Figure 6-2.4. The output is captured from a simulation run with a test bed 500 servers with a total network bandwidth of 10000. The network traffic increases during the initial phase when there is no congestion in the network. Once the traffic grows beyond 10000, inducing congestion, the network traffic begins to decline. The simulation can be terminated by raising SIGINT on the Wormulator.

```
networkTraffic = 0
networkTraffic = 35
networkTraffic = 511
networkTraffic = 891
networkTraffic = 1546
networkTraffic = 1996
networkTraffic = 3058
networkTraffic = 4489
networkTraffic = 6853
networkTraffic = 9153
networkTraffic = 11412
============ Slowing down ==========
networkTraffic = 7990
networkTraffic = 6366
networkTraffic = 4814
networkTraffic = 3499
networkTraffic = 2772
networkTraffic = 2250
networkTraffic = 1746
networkTraffic = 1438
networkTraffic = 1050
networkTraffic = 759
networkTraffic = 681
networkTraffic = 557
networkTraffic = 378
networkTraffic = 368
networkTraffic = 251
networkTraffic = 306
networkTraffic = 192
```

Figure 6-2.4 Simulation Output, where 'NETWORK_BADNWIDTH= 10000'

The worm spread is triggered by sending the UDP worm to a simulated server. This is achieved by a script whose pseudo code in given in Figure 6-2.5.

```
Create UDP socket.
IP address of the compromised machine is Ad, port Pd
Send absolute path to the worm to Ad: Pd
```

Figure 6-2.5 Initial worm spread

6.3 Collection of Data

The most important data generated during the simulation is the instantaneous network traffic which is logged under '~/home/user/Files/lnsTraffic.txt'. A sample of the instantaneous network traffic is given in Figure 6-3.1. As it can be observed the network traffic continues to increase up to 10000 (NETWORK_BANDWIDTH), after which congestion sets in and traffic begins o decline.

```
1260390926 0
1260390937 93
1260390938 578
1260390939 3134
1260390940 4422
1260390941 6245
1260390942 7903
1260390943 10332
1260390944 8731
1260390945 6807
1260390946 5289
1260390947 4066
1260390948 3209
1260390949 2667
1260390950 1994
1260390951 1688
1260390952 1384
1260390953 794
1260390954 843
1260390955 660
1260390956 524
```

Figure 6-3.1 Sample of instantaneous network traffic

6.4 Results

The data collected in section 6.3 Collection of Data is plotted using Gnuplot. Figure 6-4.1 depicts the network traffic generated by the UPD worm on a test bed of 25000 servers.

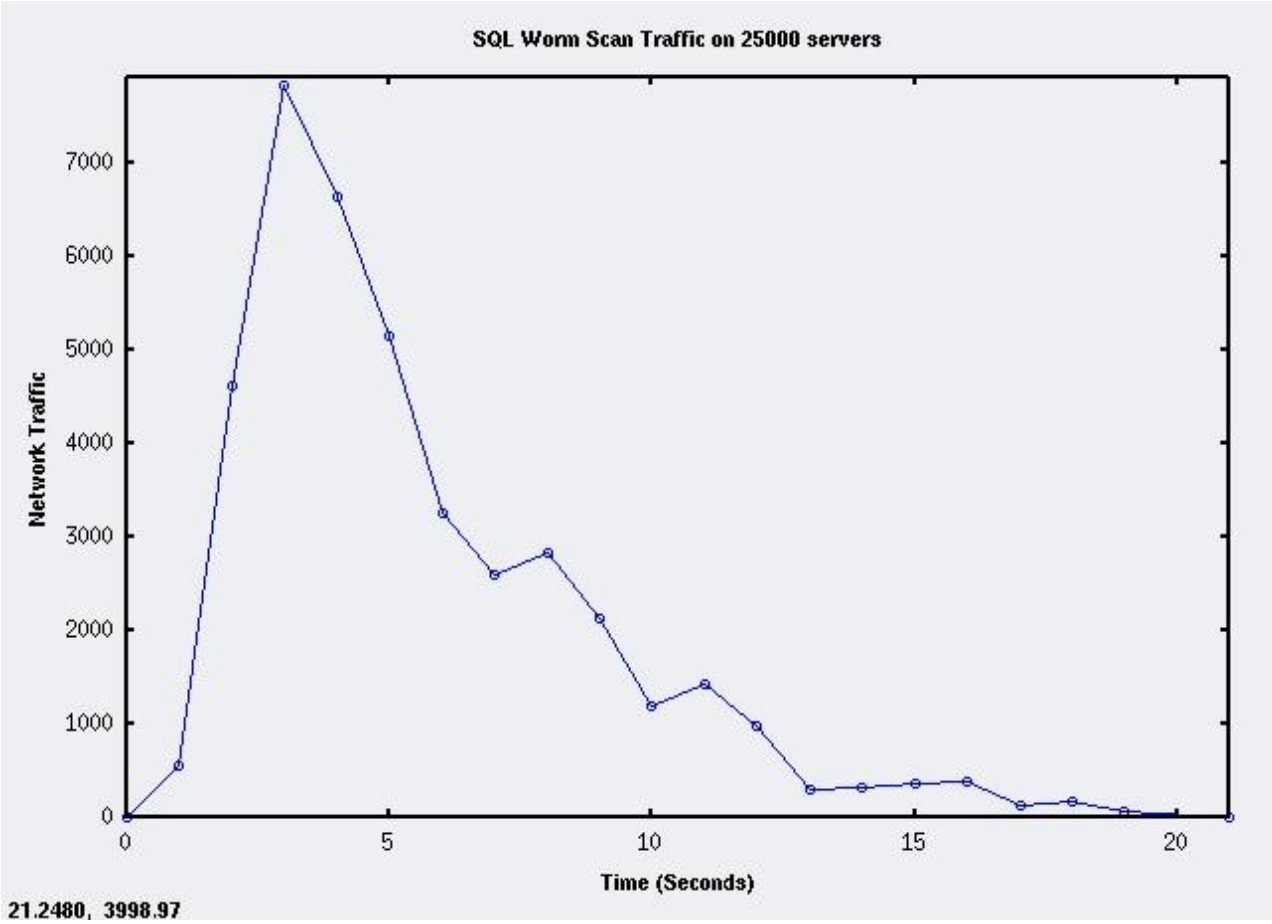

Figure 6-4.1 Network Traffic for a test bed of 25000 servers, network bandwidth = 7000

The network traffic increases almost exponentially until the limit of the network bandwidth, beyond which the traffic begins to decline due to exhaustion of bandwidth of the network. The result of our simulation is comparable to the behavior exhibited by the real SQL Slammer worm depicted in Figure 6-4.2.

The network traffic generated by the UPD worm is measured for varying test bed of nodes: 1) 1000, 2) 5000, 3) 10000, 4) 15000, 5) 20000, and 6) 30000 nodes. The simulation results are illustrated in Figure 6-4.3 to Figure 6-4.8.

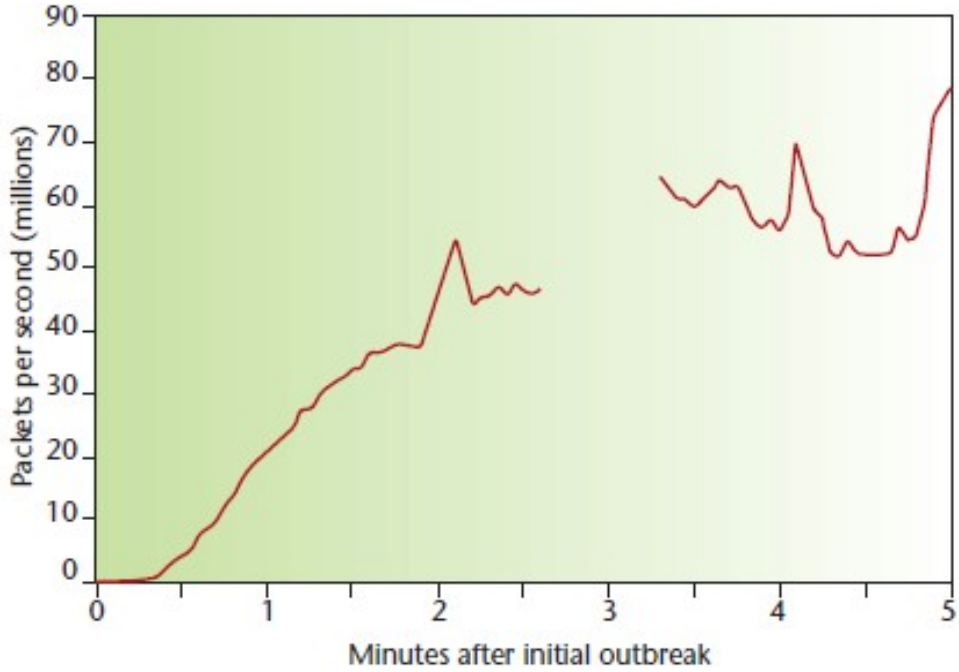

Figure 6-4.2: Slammer's early progress as measured by WAIL, tarpit [5]

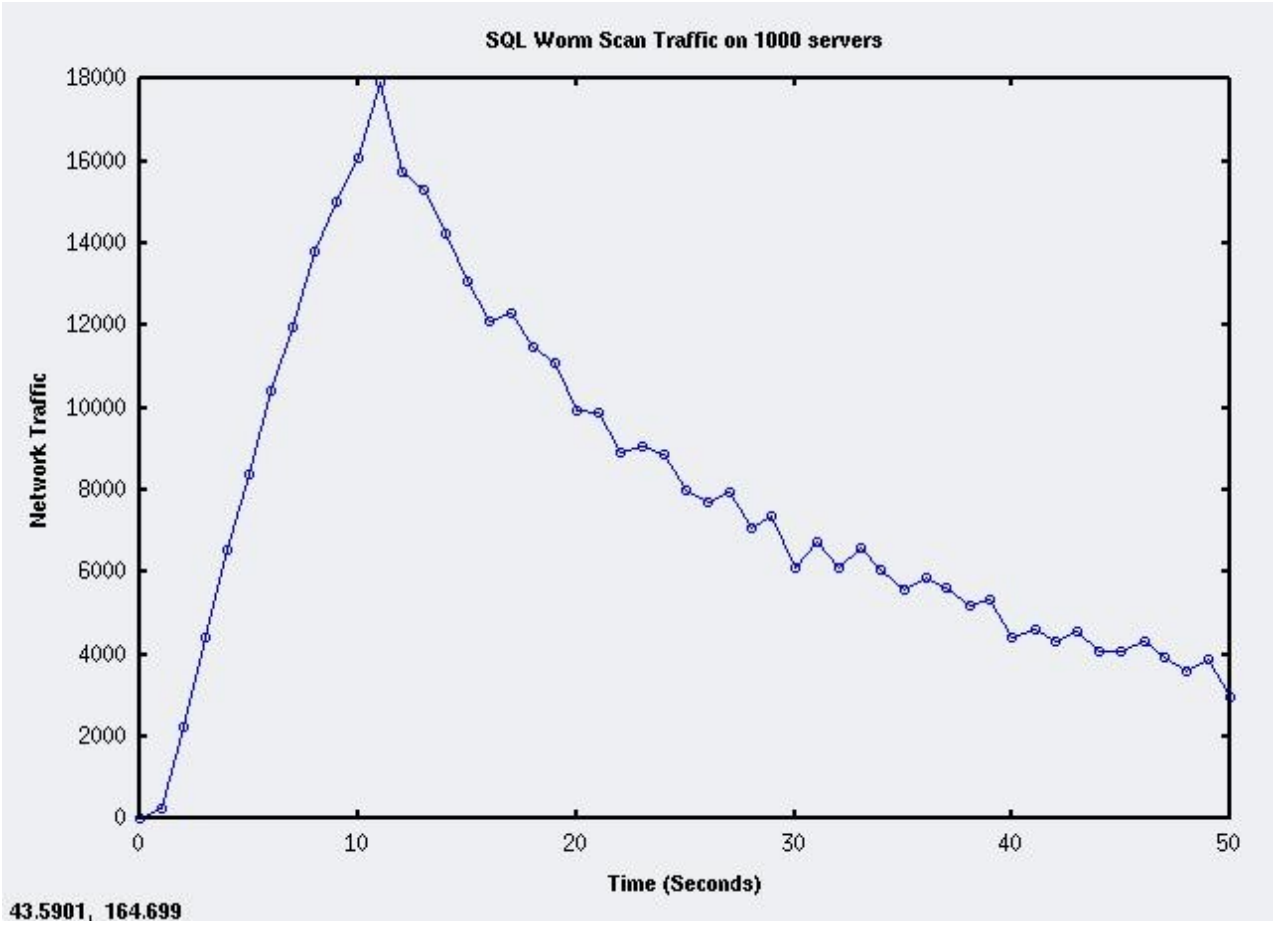

Figure 6-4.3: Simulation output for 1000 simulated servers

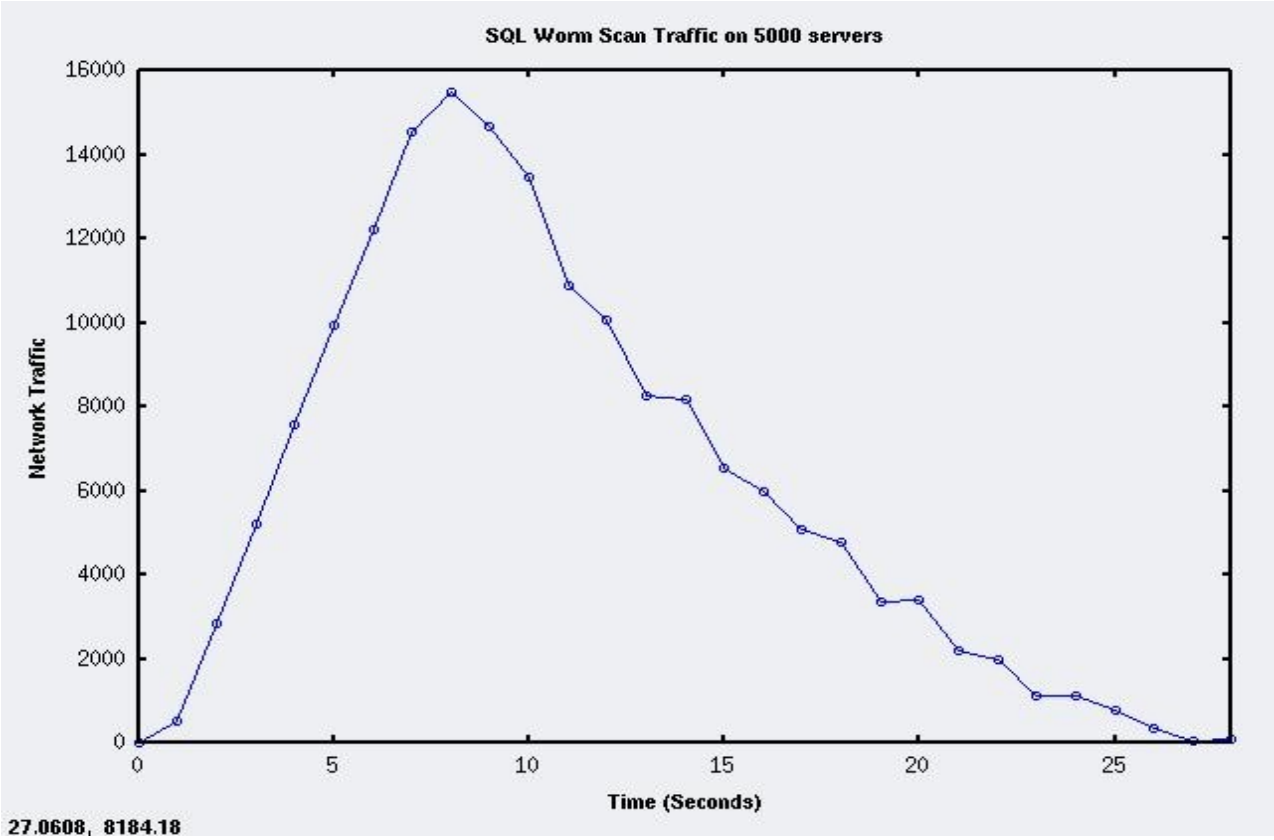

Figure 6-4.4: Simulation output for 5000 simulated servers

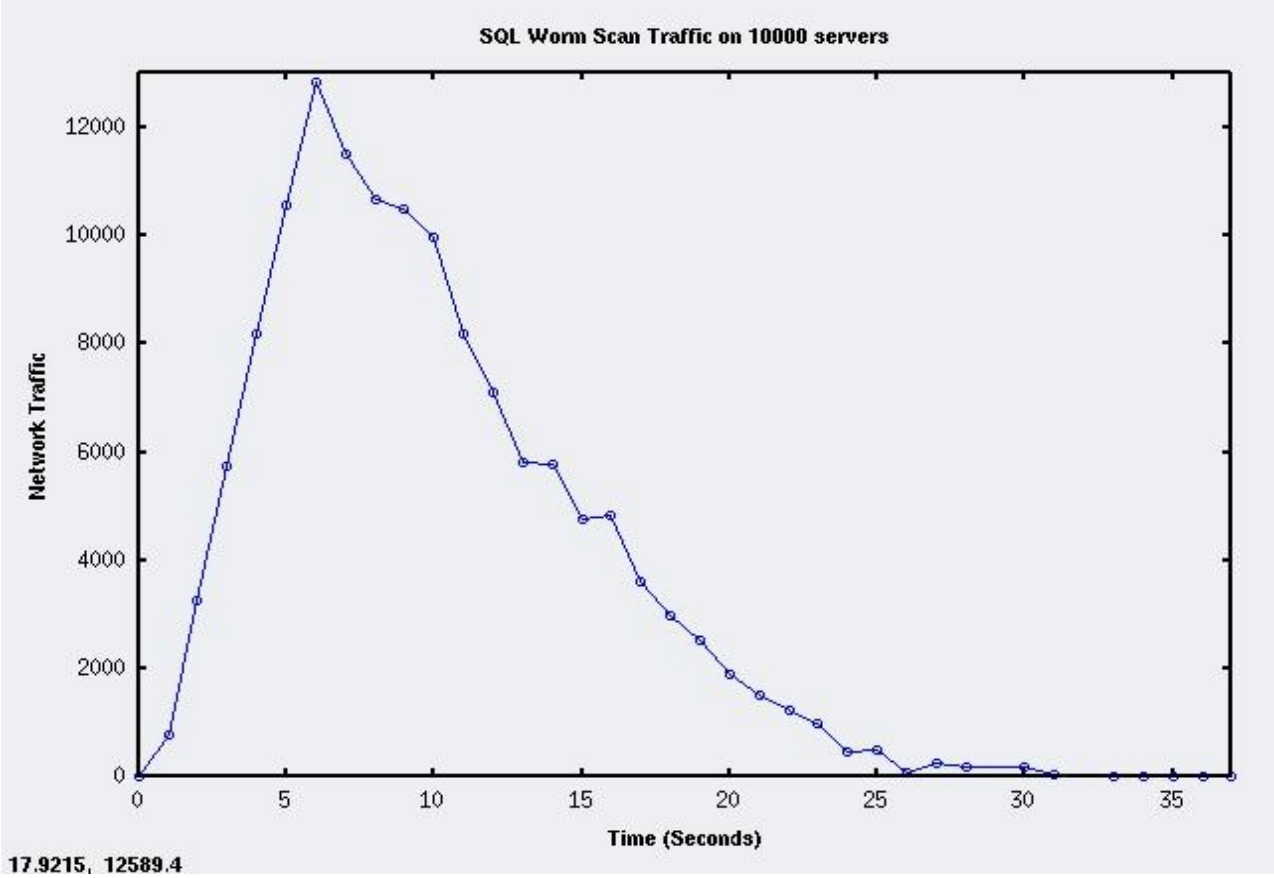

Figure 6-4.5: Simulation output for 10000 simulated servers

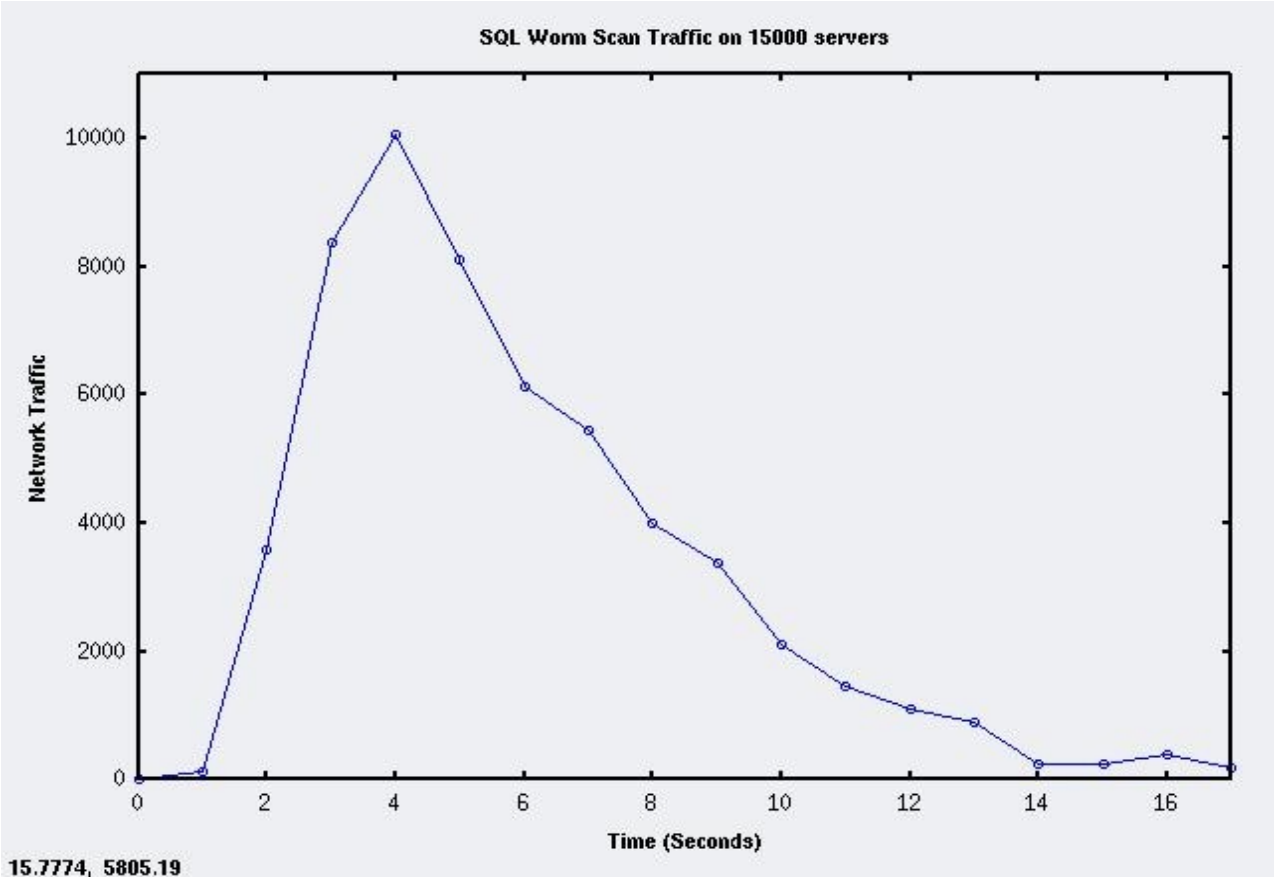

Figure 6-4.6: Simulation output for 15000 simulated servers

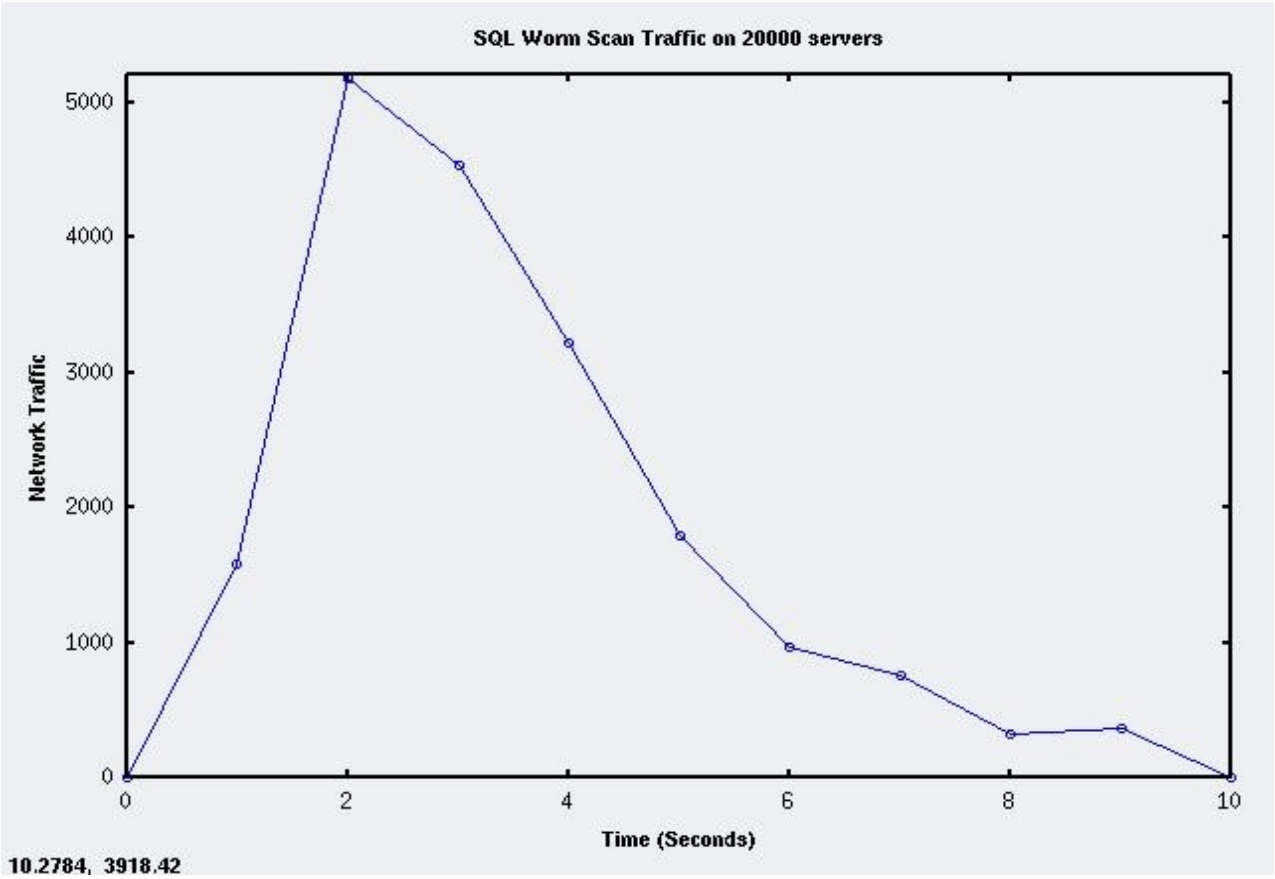

Figure 6-4.7: Simulation output for 20000 simulated servers

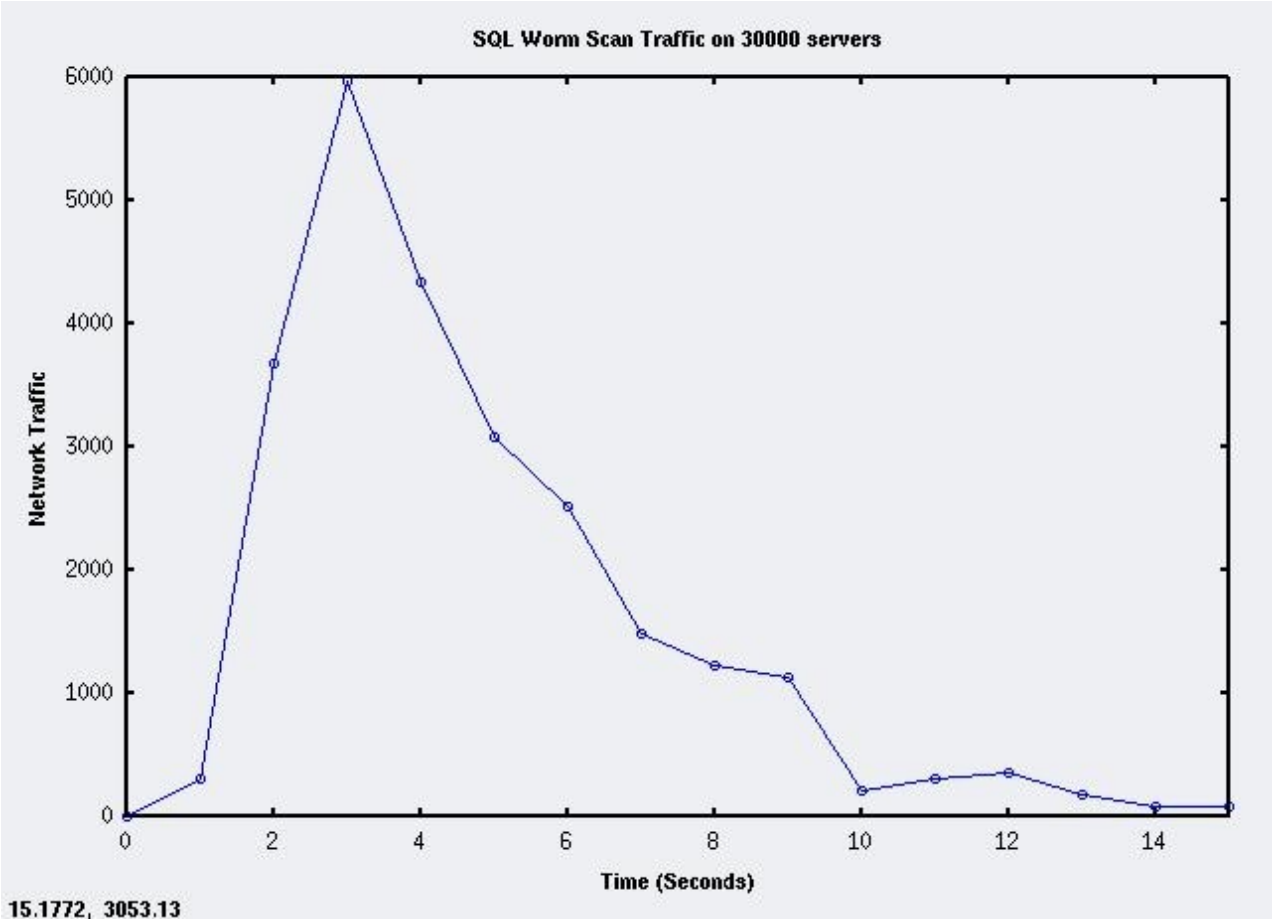

Figure 6-4.8: Simulation output for 30000 simulated servers

6.5 Evaluation of the Results

The results obtained in the above figures reveal a pattern in rate at which congestion sets in to the network. The pattern is depicted in figure 6-4.9. For a test bed of 10, 50, and 100 nodes the time to reach congestion is considerably high, greater than the number of nodes. However, when the number of nodes increases beyond 500, the speed at which congestion sets in is increases proportionately. This observation corroborates the understanding that higher the number of infected nodes, greater the worm scan traffic, which in turn exhausts the network bandwidth faster.

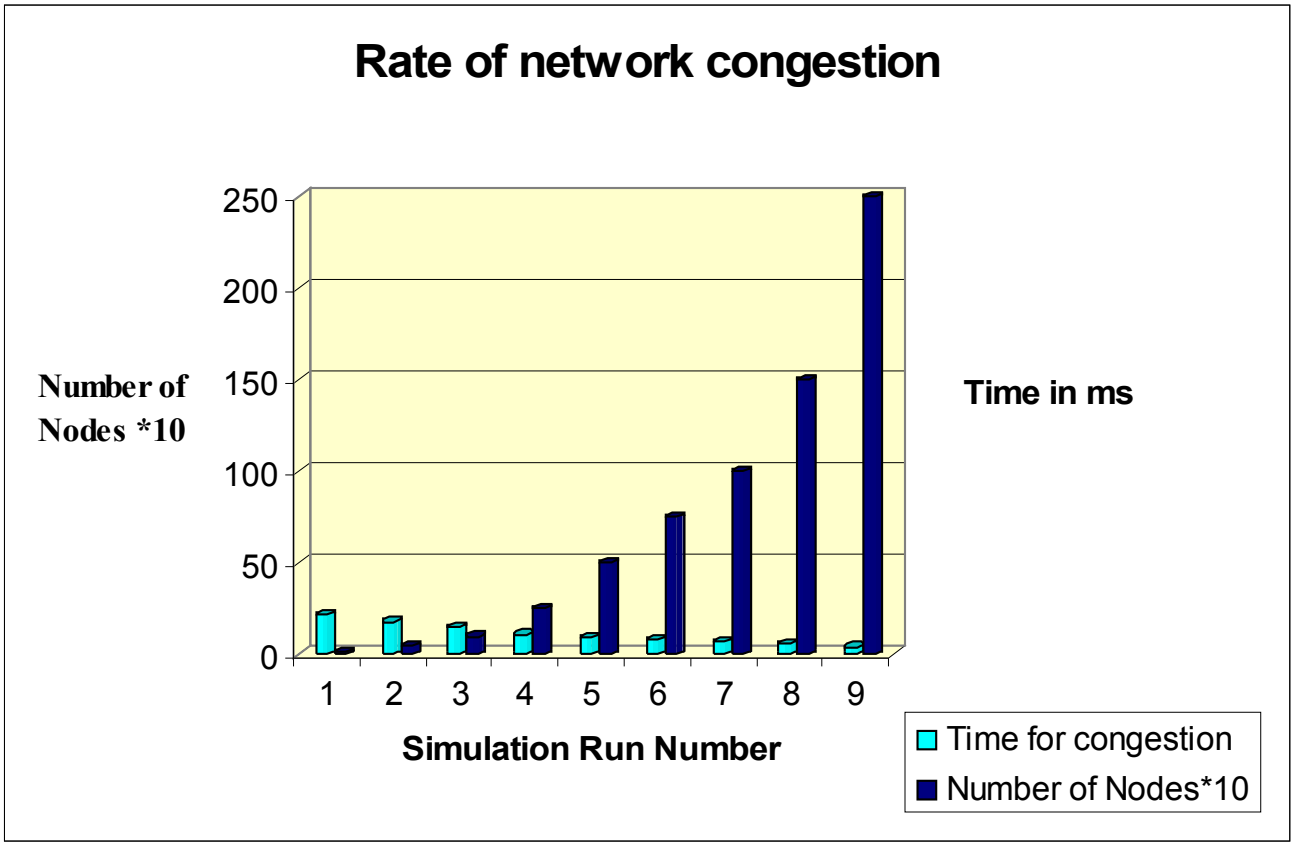

Figure 6-4.9: Rate of network congestion

The simulation scaled successfully to a test bed 30,000 servers. There was no memory available to create more threads and scale the simulation beyond this limit. The simulation as run on single processor of 1.7 Ghz and a memory of 1.25 GB. The simulation can scale further if run on a larger memory or multi processor machine.

Chapter 7

Inference

Since the beginning of this project in January 2009, several potent malware have struck the Internet all over the world. According to a Microsoft report released in November 2009, worms have almost doubled from the second quarter of 2008 to become the "most significant threat," in the first quarter of 2009 [34]. Taterf, a worm stealing login credentials for gaming, increased its presence by 156% to 4.9 million [34]. July 2009 witnessed a series of coordinated cyber attacks against major government, news media and financial websites in the United States and South Korea. The attack involved activating a botnet causing a Denial-of-Service attack [35].

To understand the development, progress and the potential risks of worms, we need to observe and study the worms and malware in a controlled environment.

7.1 Proposal

An application level worm simulator to replicate Internet-wide activities not only formalizes the behavior of malware, but it also builds a foundation for further investigation. A worm simulator can be used to observe the behavior of malware on an isolated network, project the devastating effects on the real Internet, and propose and test new defense mechanisms. The simulator will continue to evolve as our knowledge about the worms mature.

The Wormulator is developed to teach and research rapidly spreading malware in universities. It is not a full fledged network simulator. It is developed for simulating applications without the overhead of low level network simulation.

Together with the Wormulator, which implements UDP, and the Spamulator, which implements TCP, it is possible to simulate any Internet application. Even though the simulation implements only a subset of rapidly spreading malware, there are several other types of malware that can be simulated using the Wormulator.

The results obtained from this simulation project bear a qualitative resemblance to the behavior exhibited by the actual SQL Slammer worm. Wormulator provides a convenient and coherent environment to simulate Internet-wide phenomenon on a single machine. However, the aim of our simulation is not to show that the Wormulator is the absolute solution for simulating all types of malware. Additional implementations may be needed to make the Wormulator more robust and pertinent.

7.2 Future Work

The Wormulator must be updated as and when necessary for it to be accepted and widely used. This can involve significant modifications ranging from fundamental changes in the structure of classes and relationship between them, to a re-modeling a significant portion of the tool. The changes will result from the natural progression of the tool and our knowledge about malware.

The performance of the Wormulator can be improved in several other ways. Some of those include, a better error reporting mechanism, enhancements such as using standard stl::map instead of the non-standard hash map, a simple graphical user interface as an alternative to diagnostics printed on the standard output, restructuring of the modules of the Wormulator into header and implementation files, implementing network concepts such as routing and topology, and capture many other statistics such as number of infections received, whether the server is infected, etc.

The work done in this project and the results obtained can serve as the basis for the simulation of next generation malware. With little or no modifications, the Wormulator and the Spamulator can be used to simulate:

- Email and Internet Relay Chat worms that use Topological scanning.
- Flash worms that use passive harvesting of target IP addresses.
- Worms that employ subnet and permutation-based scanning techniques.
- Self modifying, metamorphic malware.
- Honey pots, SQL Injection attacks, Distributed Denial-of-Service attacks.

Testing a distributed worm simulator is a challenge, since large networks are difficult to set up and maintain. The work and the results of this project were developed and evaluated on a single processor Intel machine. A network test bed such as Emulab, provides researchers with a wide range of distributed and networked environments that can be used to develop, debug and evaluate their systems [17]. Our simulation can be run on such test beds to further validate its robustness and relevance.

Chapter 8

Conclusion

Internet worms continue to adopt ingenious methods for which there are no known defense mechanisms. In 2009 worms have been ranked as the second most prevalent security threat. According to a security report from Microsoft, the United States, United Kingdom, France and Italy are beset by Trojans. Brazil was attacked by malware targeting online banking, and Spain and Korea saw the dominance of worms led by threats targeting online gamers [34]. Study of malware and their defense mechanisms is an active field of research.

Simulation of rapidly spreading malware is the first step in our fight against security threats. The Spamulator, which supports TCP only, is not suitable to simulate UDP applications. Hence, the Spamulator is modified to create the Wormulator, which implements connectionless User Datagram Protocol. Apart from this, the Wormulator is enhanced with the ability to track instantaneous network traffic, and identify and signal congestion throughout the network. Finally, the Wormulator is enhanced further by implementing a network of nodes by creating POSIX threads instead of forking child processes. Each node functions as both, a server to receive worm packets and as a client to propagate the worm.

The effectiveness of our simulation is validated by simulating a UDP worm, the SQL Slammer worm, and evaluating it against the real world worm performance. This proof by example simulation produced an exponential growth in the instantaneous network traffic, creating congestion in the network. Moreover, the simulation scaled effectively up to a test bed of 30,000 nodes.

In conclusion, we believe that we have created a common ground to simulate all types of malware.

References

[1] 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.

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

[3] Nicholas Weaver, "Potential Strategies for High Speed Active Worms: A Worst Case Analysis," Mar 24, 2002. [Online]. URL: http://www.cgisecurity.com/lib/worms.pdf. (Accessed: Mar 30, 2009)

[4] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, Andreas Terzis, "A Multifaceted Approach to Understanding the Botnet Phenomenon," in Proceedings of the 6th ACM SIGCOMM conference on Internet measurement, Rio de Janeiro, Brazil, 2006, pp. 41 – 52.

[5] 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.

[6] 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.

[7] 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 30, 2009)

[8] Gilbert Chen, Joel Branch, Michael J. Pflug, Lijuan Zhu, and Boleslaw K. Szymanski, "SENSE: AWIRELESS SENSOR NETWORK SIMULATOR," Advances in Pervasive Computing and Networking, Springer, NY, 2004, pp. 249-267. [Online]. URL: http://cgi2.cs.rpi.edu/~szymansk/papers/wpcn.04.pdf. (Accessed: Oct 30, 2009)

[9] Erich Schikuta, "NeuroWeb: An Internet-Based Neural Network Simulator," Tools with Artificial Intelligence, (ICTAI 2002), in Proceedings of the 14th IEEE International Conference, Nov 2002, pp. 407 – 412.

[10] UCB/LBNL/VINT groups, UCB/LBNL/VINT, "Network Simulator (NS)," May 2001. [Online]. URL: http://www.isi.edu/nsnam/ns/. (Accessed: Jun 16, 2009)

[11] Siming Lin1, Xueqi Cheng and Jianming, "A Visualized Parallel Network Simulator for Modeling Large-scale Distributed Applications," Parallel and Distributed Computing,

Applications and Technologies, (PDCAT '07), in Proceedings of the Eighth International Conference, Dec 2007, pp. 339 – 346.

[12] H. Akhtar, "An overview of some network modeling, simulation and performance analysis tools," iscc, pp.344, 2nd IEEE Symposium on Computers and Communications (ISCC '97), 1997.

[13] 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, pp. 71 - 79, 2005.

[14] Dr. George F. Riley, Georgia Institute of Technology, "The Georgia Tech Network Simulator," in Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research, Karlsruhe, Germany, 2003, pp. 5 – 12.

[15] Xiang Zeng, Rajive Bagrodia, Mario Gerla, "GloMoSim: a Library for Parallel Simulation of Large-scale Wireless Networks," in Proceedings of the 12th Workshop on Parallel and Distributed Simulations (PADS '98), May 1998, Banff, Alberta, Canada.

[16] Henri Casanova, Arnaud Legrand and Martin Quinson, "SimGrid: A Generic Framework for Large-Scale Distributed Experimentations," in Proceedings of the 10th IEEE International Conference on Computer Modeling and Simulation (UKSIM/EUROSIM'08). [Online]. URL: http://www.loria.fr/~quinson/articles/SimGrid-uksim08.pdf. (Accessed: Oct 14, 2009)

[17] Emulab, last accessed 17 Dec 2007. [Online]. URL: http://www.emulab.net. (Accessed: Nov 3, 2009)

[18] B. Kneale, A. Y. De Horta, I. Box, "VELNET (Virtual Environment for Learning Networking)," in Proceedings of the 6th Australasian Computing Education Conference(ACE2004), Dunedin, New Zealand, v. 30, pp. 161–168

[19] Oskar Andreasson, "Iptables Tutorial," Version 1.2.2, 19 Nov 2006. [Online]. URL: http://iptables-tutorial.frozentux.net/iptables-tutorial.html. (Accessed: Dec 3, 2008)

[20] Linux Programmer's Manual (3)

[21] Mani Radhakrishnan and John Sloworth, "Socket Programming in C/C++," Sep 24, 2008. [Online]. URL: www.rites.uic.edu/~solworth/sockets.pdf. (Accessed: Feb 15, 2009)

[22] A joint effort of CAIDA, ICSI, Silicon Defense, UC Berkeley EECS and UC San Diego CSE, "Analysis of the Sapphire Worm," Feb 12, 2007. [Online]. URL: http://www.caida.org/research/security/sapphire/. (Accessed: Mar 5, 2009)

[23] David M. Nicol, "Efficient simulation of Internet worms," ACM Transactions on Modeling and Computer Simulation (TOMACS), v 18 , Issue 2 , Article No. 5, Apr 2008.

[24] Brian Beej Jorgensen Hall, "Beej's Guide to Network Programming Using Internet Sockets," version 3.0.13, 2009. [Online]. URL: http://beej.us/guide/bgnet/. (Accessed: May 23, 2009)

[25] C.Col´on Osorio, Zachi Klopman, "An Initial Analysis and Presentation of Malware Exhibiting Swarm-Like Behavior," in Proceedings of the 2006 ACM symposium on Applied computing, Dijon, France, pp. 323 – 329.

[26] Kalyan S. Perumalla, Srikanth Sundaragopalan, "High-Fidelity Modeling of Computer Network Worms," in Proceedings of the 20th Annual Computer Security Applications Conference, pp.126 - 135, 2004.

[27] Norman Matloff, "Overview of Computer Networks," April 11, 2005. [Online]. URL: http://heather.cs.ucdavis.edu/~matloff/Networks/Intro/NetIntro.pdf. (Accessed: Jan 3, 2009)

[28] Songjie Wei and Jelena Mirkovic, "A Realistic Simulation of Internet-Scale Events," in Proceedings of the 1st International Conference on Performance Evaluation Methodologies and Tools, Pisa, Italy, Italy Article No. 28, 2006.

[29] Sven Goldt, Sven van der Meer, Scott Burkett, Matt Welsh**,** "The Linux Programmer's Guide," Version 0.4, 1995. [Online]. URL**:** http://tldp.org/LDP/lpg/lpg.html. (Accessed: Feb 13, 2009)

[30] Blaise Barney, "POSIX Threads Programming," Last Modified: 07/27/2009. [Online]. URL: https://computing.llnl.gov/tutorials/pthreads/#Overview. (Accessed: Mar 29, 2009)

[31] David A Rusling, "The Linux Kernel," Version 0.8-3, 1999. [Online]. URL: http://tldp.org/LDP/tlk/ipc/ipc.html. (Accessed: Aug 8, 2009)

[32] S. Staniford, V. Paxson, N. Weaver, "How to Own the Internet in Your Spare Time," in Proceedings of the 11th USENIX Security Symposium, USENIX, pp.149 – 167, Aug 2002.

[33] J. Nazario, J. Anderson, R. Wash, C. Connelly, "The Future of Internet Worms," Crimelabs Research, Jun 2001.

[34] Article featured in Malware and Hardware Security magazine, 02 November 2009. [Online]. URL: http://www.infosecurity-magazine.com/view/4934/information-security-threats-in-h1-2009-malware-and-rogue-security-software/. (Accessed: Nov 10, 2009)

[35] Wikipedia contributors, “Cyber Attacks,” Wikipedia, The Free Encyclopedia, Jul 2009. [Online]. URL: http://en.wikipedia.org/wiki/July_2009_cyber_attacks. (Accessed: Oct 26, 2009)

Appendix A

IP Tables and libIPQ

This section provides background information on important elements such as IP Tables and library libIPQ used in a loop back network simulator such as the Spamulator and the Wormulator.

IP tables are inherent to a loop back network simulator. IP Tables such as IP filter, IP mangle, etc., operate mainly in layer 2, of the TCP/IP stack. IP Tables are made up of a table and a chain and are applied on individual packets. Each table has a specific purpose, and there are four tables, namely Raw, Nat, Mangle and Filter tables. Each chain contains a set of rules that are applied on packets that traverse the chain. There are four types of chains, namely Pre-routing, Post-routing, Input, Output and Forward [19]. The Spamulator and the Wormulator modifies only the "MANGLE" IP Table. When a packet enters the local host, the packet comes into eth0 or eth1. The following actions take place: 1) the REROUTING chain of the IP Table Mangle, if set, mangles the packet before routing decision is made, 2) routing decision is made as to whether the packet is destined for the local machine or needs to be forwarded, 3) INPUT chain of the IP Table Mangle, if set, mangles the packet before the packet is sent to the machine, and 4) Packets are sent to the application or forwarded to another machine. Similarly, when a packet is sent from our LOCALHOST, the packet is processed before it is sent out of the machine. The OUTPUT and POSTROUTING chain of the mangle table is used instead. An example rule: "`iptables -t mangle -I OUTPUT -p udp -d 10.0.0.0/8 --destination-port 1400:1500 -j QUEUE`":

- IP table under consideration is "Mangle" and -I option inserts a rule somewhere in the OUTPUT chain.
- UDP packets destined for any IP address in the range 10.0.0.0 to 10.0.0.8 and port ranging from 1400 to 1500 is captured on a local ip_queue.

LibIPQ is a development library for iptables user space packet queuing. Netfilter in Linux provides a means of retrieving packets from the network stack and queue them to the user space. These packets maybe altered in the user space and then sent back to the kernel. Kernel module called queue handler is registered with Netfilter to pass packets to and from the user space. The queue handler for IPv4 is ip_queue. Using appropriate IP Table Rule along with the target set to QUEUE, the packets can be sent to the ip_queue module, which will then attempt to deliver the packets to a user space application. If no user space application is waiting, the packets will be dropped [20]. An application in the user space may access these packets via the libIPQ library which provides the APIs for communicating with ip_queue. For an application to use this library we need to include: 1) linux/netfilter.h, and 2) libipq.h. The following are some of the library function calls used in the Spamulator:

- `ipq_create_handle`, initializes library and returns context handle.
- `ipq_set_mode`, sets the queue mode to copy either packet metadata, or payloads and metadata to user space. It is also used to initially notify ip_queue that an application is ready to receive queue messages.
- `ipq_read`, waits for a queue message to arrive from ip_queue and read it into a buffer.
- `ipq_message_type`, determines the message type in the buffer.
- `ipq_get_packet`, retrieves packet message from the buffer.
- `ipq_get_msgerr`, retrieves an error message from the buffer.
- `ipq_set_verdict`, sets a verdict on a packet, optionally replacing its contents.
- `ipq_destroy_handle`, destroys context handle and associated resources [20].

Appendix B

Inter Process Communication

Various Inter-Process Communication mechanisms of the Linux operating system are used in the loop back network simulator as well as our simulation. This section will explore some of the commonly used IPC mechanisms, starting with network sockets, followed by unnamed pipes and ends with signals.

Network sockets provide connection between processes. Sockets can be classified as connection oriented or connectionless, packet based or stream based, and reliable or unreliable. Sockets are used for connection based client server model where the server waits for connection from the client. Sockets support different domains, types and protocols [21]. Some of the commonly used socket APIs in the simulation are:

- Create a socket of domain, type and protocol.
- Bind the address of the socket on to a server port.
- Send/receive packets.
- Shutdown and close a socket to stop reading/writing and release kernel data structures.

Less commonly used auxiliary socket APIs are: htons, htonl, ntohs, ntohl, which are used for network byte ordering. Sparc is big endian and Intel is little endian. The simulation is run on an Intel machine with Linux Operating System. Typically, a server application is bind to a particular port on the machine. The ports are defined as follows: 1) 0 – 1023 port numbers can be used only by root, 2) 1024-5000 port numbers used by popular applications, and 3) 5001-64K port numbers are ephemeral ports used by user defined applications. The real SQL Slammer worm attacked port 1434 of a MS SQL server. However in our simulation, since all servers run on the same the localhost, 127.0.0.1, each server is bind to a unique port number starting from 14000.

Pipes are unidirectional byte stream, which connect the standard output of one process to the standard input of another process. When a process creates a pipe, the kernel sets up two file descriptors. One descriptor is used to write data into the pipe, while the other is used to read data from the pipe [29]. A pipe in our simulation is created to communicate between the Wormulator and a child server process. Since a pipe is unidirectional, the Wormulator closes the write end and keeps the read end open. Similarly, the child server closes the read end and keeps the write end open. When the server writes to the standard output, the data is redirected to the Wormulator. The server writes its port number in this manner.

Signals are used to signal asynchronous events to one or more processes. Signals can be generated by a keyboard interrupt, generated by the kernel or other processes in the system [31]. The signals used in the simulation are:

- `SIGINT`, raised by the keyboard interrupt is handled by the Wormulator, which will terminate all its child server processes and itself.
- `SIGALRM`, is a timer which terminates an execution after the timer runs out. This is used by the Wormulator while it waits for a server to write its port. If the server does not respond within the time limit, the server connection is terminated.
- `SIGCHLD`, signals the termination of a child process to the parent process. The Wormulator handles this signal, which is raised when one of its child server processes exit.
- `SIGIO`, is raised when the child server process writes into the pipe created between the Wormulator and itself. This is raised when the server writes its port number to the pipe.
- `SIGUSR1` and `SIGUSR2`, are user defined signals. In our simulation, these signals are raised by calling the kill() system call in the Wormulator and are handled in each of the child server processes. SIGUSR1 signals congestion free network and enables the worm to grow, and SIGUSR2 signals congested network and inhibit the worm's growth.

Appendix C

POSIX Threads

This section discusses some of the multi threading techniques used in the simulation. In Linux everything is treated as a thread by the Kernel. By definition, a thread is a light weight process that exists within the main process and shares the process resources. However, each thread has its own registers, thread specific data on its own stack, scheduling priorities, and signals [30]. A thread can be independently scheduled by the kernel, involves less overhead in creation and maintenance. As a result, CPU context switching between threads is faster.

The Wormulator launches a server by creating a separate child process, using the fork() system call. The server is executed in this child process. The UDP worm in our simulation generates random IP server address and sends itself, in an infinite loop. It is imperative that the server, which receives the worm packet, does not get blocked executing the worm code. Therefore, a thread is created, which will execute the worm code. The following POSIX pthread functions are used in our simulation:

- `pthread_create` creates a new thread and makes it executable. In our simulation the server creates a thread when it receives a worm packet for the first time. The worm code is executed in this thread. No threads are created for subsequent infections of the server.
- `pthread_exit` is used to explicitly exit a thread. This function is called when the server receives a SIGINT from its parent process, the Wormulator.