A Virtual Testbed to Evaluate Worm Defense Techniques

by

Shuang Hao

B.Sc., Tsinghua University, 2002
M.Sc., Tsinghua University, 2005

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

October, 2007

Abstract

The rapidly growing amount of malicious software (such as worms) on the Internet causes significant security problems in enterprise networks and has been attracting increasing research attention. Many methods have been proposed to detect, throttle or even prevent the spreading of malware. However, most of the research experiments to evaluate the effectiveness of these defense mechanisms are based on off-line testing, synthetic data or mathematical modelling, which are unable to convincingly validate the efficiency of the defense systems. Better evaluation testbeds with live worms mixed with realistic traffic are required to help facilitate research on malware defenses.

In this thesis we focus on developing a testbed which provides an emulation of realistic traffic conditions for network and security researchers. The system is constructed using virtual hosts, which makes the testbed scalable and flexible. Network traffic is collected from a real enterprise network and then replayed in the virtual environment. In the meantime, vulnerable services on the virtual hosts allow actual malware to compromise individual hosts and flood the virtual network.

Our use of virtualization technology enables an all-software implementation. It grants fast and convenient generation, startup and shutdown of the testbed. The data-link layer virtualization and the port-based forwarding VLAN strictly confine the released malware within the testing environment. The virtual smart switches provide a platform for researchers to evaluate the security and usability of their protection architecture against worms.

Table of Contents

List of Figures

Acknowledgements

I would like to express my gratitude to all those who have offered me help in completing this thesis. Especially, I owe the greatest thanks to my supervisors, Dr. William Aiello and Dr. Norman Hutchinson, who provided me with excellent guidance and support in the entire process of this thesis project. I want to thank Dr. Andrew Warfield for giving me insightful comments on this work, and being my second reader.

I would like also to thank all the members of the system lab for their constructive suggestions. Without their help, this work would not be done. I thank all my friends at the University of British Columbia, who make my life and study at Vancouver really happy.

Most importantly, I want to thank my family from the bottom of my heart. They always encourage me to pursue my dream and their support is the power for me to finish this thesis.

Chapter 1

Introduction

1.1 Motivation

Worms pose a significant threat to existing and future networking infrastructure, as they can infect hundreds or thousands of hosts in a very short period of time. Malicious mobile codes, especially worms, have attracted increasing attention. It cost a fortune to cleanse the infected hosts and the tainted networks. Especially infected enterprises suffer much from the malicious behaviors of worms, since the LANs (Local Area Network) are complex and the operating systems and software on desktops are usually diverse. The breakout of a worm in an enterprise can cause millions of dollars of damage. Moreover, active worms can potentially spread across the Internet within seconds [30], therefore manual detection and prevention is impossible. Many automatic methods have been proposed to identify and control the spreading of worms. The most dangerous worms are those newly released ones, since at that time the anti-virus community knows little about which vulnerability the worms would exploit and what payload the worms carry. In order to deal with novel worms, people have been trying to analyze the network behaviors of the worms.

Providing a defense against malicious software requires us to walk a fine

line between security and usability. A security policy may be so strict that it adversely effects the usability/availability of the system. The extreme case is when all packets on the network are dropped, thus no attack attempt is possible. However, this is equivalent to the network being cut off. An event, incorrectly identified by the defense system as being an intrusion when none has occurred, is called a false positive [13]. On the other hand, a relaxed security policy may be unable to detect the malicious traffic. For example, if it allows all the traffic through, the policy will guarantee the usability of the network, but the worms are free to spread in the network. A false negative [13] is an event where the defense system fails to identify an attack when one has in fact occurred. Due to the complexity and diversity of the systems and networks, it is impossible to build a flawless system or design omniscious anti-worm software. Therefore researchers are continuously seeking a good tradeoff between security and usability, i.e., struggling to detect the new worms' spreading in their early stages while doing little harm to legal communication in the network.

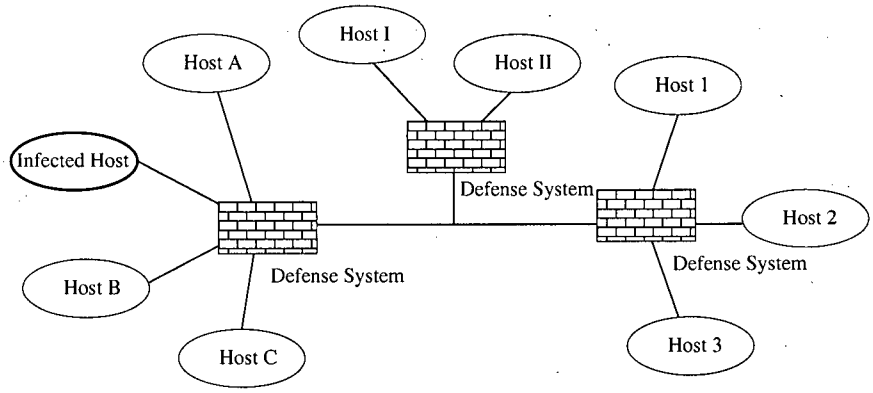
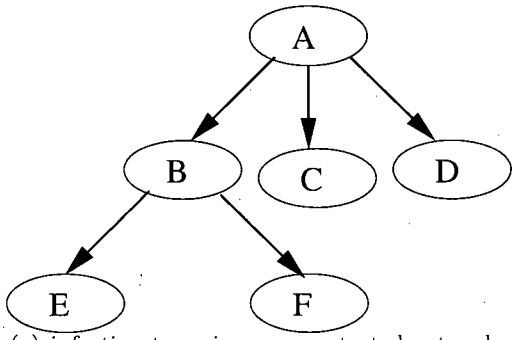


Figure 1.1: An example testbed for worm research

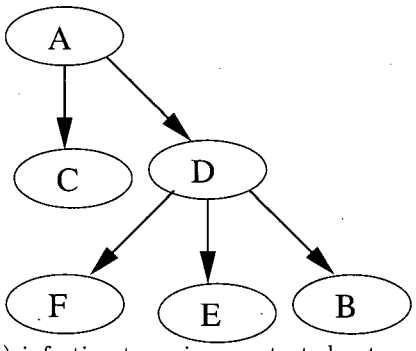
A typical testbed setup is shown in Figure 1.1. The hosts indicated by English characters, numbers and Roman numerals belong to different network segments respectively. The defense systems isolate those parts of the network. When an infected machine (indicated as the bold ellipse) sends out packets carrying a malicious payload, the defense systems are expected to detect the abnormality of the traffic and deploy the appropriate response. Afterwards, the dropped and allowed packets could be counted to measure the accuracy of the defense systems. Security researchers would like testbeds that contain authentic network traffic and provide a flexible playground to investigate the effect of the defense systems. A lot of mathematical models have been proposed to characterize the propagation of worms [18, 21, 30]. They provide a theoretical view to help us understand the worms' nature and defend against their spreading. However, those models are not very suitable to evaluate the defense algorithms, since the concrete worms might behave differently depending on the exploited vulnerabilities and running hardware systems. Another problem for testing on the synthetic worms' propagation models is their lack of actual background traffic. People hope to accurately test the rates of false positives and false negatives with the mixture of the worms' spreading and the normal traffic. Many platforms, such as Emulab and PlanetLab, have been constructed to facilitate network and system research. Although they provide full system privilege and easy network configuration, the security experiments might cause chaos if real worms were released in the environment and the normal connections created by the users do not reflect the actual traffic in real enterprise LANs.

An alternative scheme is to do the evaluation offline. Background traffic

and worm propagation could be collected separatedly and the two sets of data can be mixed later for test. However, during the evaluation, the actions of the defense system will break the recorded infection trace.



(a) infection trace in an unprotected network



(b) infection trace in a protected network

Figure 1.2: Infection trace tree

Suppose the infection trace we collect from an unprotected network is as shown in Figure 1.2 (a). A is the seed infected machine. Then B, C, D are compromised (at time t_1). Consequently B will infect E and F next (at time t_2). If the attack from A to B is successfully blocked by the defense methods, then it is hard to estimate when B and its subnodes E and F will be compromised in the environment with the defense system deployed. At

least B will not be infected at time t_1, and maybe later the 3 machines will all be attacked by machine D as shown in Figure 1.2 (b). This example demonstrates that the nodes' states and malicious traffic in a protected network are difficult to be reconstructed from those in an unprotected network. Therefore offline evaluation is unappropriate to test the anti-worm defense mechanisms.

Our goal is to construct an experimental testbed, where actual worms will be released and mixed with the background traffic sampled from a real enterprise LAN. Our intention is to provide security researchers an effective platform to investigate the security vs. usability of tradeoff in the anti-worm defense mechanisms. A testbed with authentic traffic and real vulnerable systems will surely facilitate the development of research against worm attacks.

1.2 Contribution

In this thesis we explore constructing such a security testbed, based on the technique of virtualization, i.e., an all-software implementation. The released worms have the illusion that they reside in the actual vulnerable systems and propagate on the real network.

Obviously, people could build up a testing sandbox with several physical machines. However, the scheme of using direct physical machines is fundamentally hard for several reasons:

- The experiments suffer from the hardware restrictions. Most changes in the experimental configuration will place extra requirements on the

hardware. For example, if a machine is set to connect multiple network segments, we have to open its box and manually install the corresponding number of network devices;

- The scale of the experiment is limited by the number of available physical nodes. It is no problem to build up a network with 10 or 20 machines, but a large network with hundreds of nodes is hard to construct;

- Once the machines are infected, it is a time-consuming work to cleanse the systems of worms. In the worst case, we might need to reinstall the systems to start a new experiment;

- There are two monitoring modes for the experiment. 1) Users monitor the experiment at the isolated testbed network, i.e., to operate on the individual test nodes. Such setup brings no security problems; 2) In the case where the monitor is not collocated in the isolated testbed network, a network connection is required out of the testbed. This configuration substantially increases the difficulty of guaranteeing that the malware remains contained within the testbed.

In order to overcome the problems listed above, this thesis represents an initial step toward the construction of an environment for safely evaluating active defenses against Internet worms in enterprise networks. Our prototype system uses a single physical host running the Xen virtual machine monitor to emulate a medium-sized enterprise network containing tens of hosts. Our system allows live, recorded background traffic to be replayed

with high-fidelity while live attack traffic is issued against hosts in virtual machines. Our work makes the following contributions:

1. We build a complete system in which hosts under attack are isolated in virtual machines on a completely virtualized Layer 2 network. This is a simple, architectural solution to the containment of malicious traffic.

2. Taking advantage of previous work on the time dilation of virtual machines, we demonstrate that background traffic, recorded from a real enterprise network, can be replayed with very high fidelity, and that as a consequence our testbed forms an excellent base for achieving repeatable results in testing defense mechanisms.

Meanwhile, our design is also aiming to achieve the following potential features in the future, and we discuss those rough ideas in the thesis.

- The testbed is easy to configure. The different system images could be loaded to boot hundreds of virtual hosts within a few minutes. After the experiment is finished, the tear-down of the virtual hosts will automatically remove the worms too;

- The network can contain smart virtual switches which are designed to run the defense algorithms under test. An arbitrary number of virtual NICs can be assigned in each smart switch to connect the virtual end hosts;

- The worms are strictly confined within the testing environment. The virtual environment is like an isolated island to the physical machines.

Moreover, it is possible for users to remotely control the testbed, while the experimental traffic is unable to leak to the outside world.

1.3 Organization

This thesis presents the design of our testbed system and the technical details of its implementation.

The remainder of the thesis is organized as follows. Chapter 2 presents a survey of related work and introduces the background knowledge. In Chapter 3, we state the basic idea and structure of the testbed for validating the enforcement architecture. We further demonstrate the detailed implementation in Chapter 4. Chapter 5 contains the experimental evaluation of our testbed. Finally we draw conclusions in Chapter 6 with some discussions of future work.

Chapter 2

Background & Related Work

Malicious mobile codes are currently prevalent on the internet, which is a problem for both individuals and enterprises. Worms are notorious among various kinds of malicious mobile codes, due to their wild breakouts. The existing worms have been carefully studied by security researchers, and many defense mechanisms have been proposed to protect the computers and networks. However, it is still an unsolved problem to stop the zero-day worms. There exist several testbeds for researchers to investigate worms' behaviors or to evaluate the counter-worm methods. In this chapter, we will briefly introduce some representative worms and the mitigation techniques against them.

2.1 Worms and Their Behaviors

A computer worm is a program that self-propagates across a network exploiting security or policy flaws in widely-used services [33]. A distinguishing feature of worms is that they spread and compromise machines without human interaction, which makes them propagate very quickly on the network and allows them to easily infect hundreds or even thousands of vulnerable machines. Typically, a worm's life cycle includes the following phases.

- Probing: In order to infect machines in a network, a worm needs to first identify the existence of other machines. The simplest way is to randomly scan the IP addresses. Even though the randomly created addresses might not be assigned to any machine or the scanned machine may not be vulnerable to the worms, the automatic activation still makes the worms spread very quickly. Recently, some worms begin to utilize localized scanning, i.e., scanning the addresses in the same network segment with high probability and scanning randomly otherwise. There exist several potentially highly virulent scanning techniques [30], including hit-list scanning, where the worm authors collect a list of IP addresses beforehand and the worms will first try to attack the listed addresses and permutation scanning, where a worm is able to detect that a host is already infected and will not continue to scan the duplicated addresses.

- Exploitation: Once the worms detect a machine running vulnerable services, usually they will launch overflow attacks (e.g., the attack methods of the first worm Morris and recent CodeRed [8]) to compromise the machine. If the programmers did not make careful bounds checking, the worms are able to override part of the normal codes and make the program execute the malicious instructions. Then the innocent machine will be infected by the worms.

- Propagation: A compromised machine will continue to infect others. Some worms need human interaction to be activated, e.g., Nimda[7] is activated after the machine is rebooted by the users. As the worms'

propagation depends on users' behavior, the spreading speed is comparatively slow. The fastest worms are self-activated, such as CodeRed [8], which starts to spread malicious codes as soon as it exploits a vulnerable machine.

- Attacks: Worms are able to execute malicious tasks on the infected machines, including theft of private information, installing backdoors, launching DDoS attacks, spreading spam etc.

Even though researchers attempt to design secure systems, there still exist a lot of potential vulnerabilities within current operating systems and software, which presents exploitation for future worms. The rapid development of networks (including their bandwidth and scale) aggravates the propagation of worms.

On July 19, 2001, more than 359,000 computers connected to the Internet were infected with the CodeRed (CRv2) worm in less than 14 hours. The cost of this epidemic, including subsequent strains of CodeRed, is estimated to be in excess of $2.6 billion [27]. It will exploit a buffer-overflow vulnerability in Microsoft's IIS web servers. A backdoor will be installed once a machine is infected by CodeRed. With probability 1/8, CodeRed probes random IP addresses; the rest of the time scanning the local network with the same class A or B addresses.

Slammer (a.k.a. Sapphire) was the fastest computer worm in history. As it began spreading from 05:30 UTC on Saturday, 25 January 2003, the worm infected more than 90 percent of vulnerable hosts within 10 minutes [26]. It simply sends a packet to UDP port 1434 to exploit the buffer-overflow

vulnerability on SQL servers. Despite the fact that Slammer also adopts a random scanning strategy, its one-packet attack and small size (404 bytes) allows fast spreading speed.

2.2 Worm Mitigation Techniques

The outbreaks of worms cause much trouble to both individual and enterprises machines and cause significant networks congestion. Security researchers pay a lot of effort to study techniques against worms and many defense mechanisms have been proposed to identify and control their spreading. Roughly they can be divided into 3 strategies: Proactive Protection, Reactive Defense and Local Containment [17].

Proactive protection aims to protect the system by reducing the possibility for a worm to exploit a given vulnerability. Such methods include sandboxing, privilege separation, system call monitoring, etc [17]. Regardless of specific worms' exploitation, proactive protection tries to enhance the overall security. For example, since most worms take advantage of unchecked buffers, an effective counter method is to randomly change the addresses of the stack or heap, which will make the worms confused about the internal states of the machines [16, 19, 37]. Proactive protection sometimes completely block the worm attacks, but it is impossible to construct a system or network without any vulnerabilities.

Reactive defense needs specific information to prevent worms. Against the known worms, security patches will be applied to eliminate the vulnerability. The defense systems also try to detect the anomaly in the packets'

content to filter the malicious traffic. If the legitimate states can be clearly specified, reactive defense might be able to stop the novel worms' spreading.

Local containment treats each individual machine as a potential suspect rather than a victim. The defense systems monitor the outgoing traffic and prevent the infected machine from attacking the rest of the network. The throttling schemes do not focus on extinguishing the worms, but on greatly slowing their spreading speed [25, 31, 36]. The efficiency of throttling highly depends on the deployment ratio [17]. If the defense systems can be ideally deployed in the entire network, the outbreak of worms are able to be greatly contained.

2.3 Other Evalation Testbeds

There exist several limits that encumber the evaluation for security mechanisms.

- Lack of representative data: The traffic, topology and protocols on the networks are too complicated to construct reasonable artificial data. On the other hand, few enterprises or ISP share their data with the public, due to privacy concerns.

- Deficiency of malware simulation: Researchers have tried to validate the security mechanisms by mathematical models of the worm spreading. However, theoretical analysis is not convincing from the system view and the evaluation results are sometimes vague.

- Problem of evaluation in the real world: Evaluation can be processed

in the real network with authentic traffic and malware spreading (if there exists any). The problems of such test include 1) The unmature defense system might hinder the legitimate traffic; 2) The testing procedure can not be repeated, which makes the evaluation results hard to be verified; 3) The test network might not have target worms, and the malicious codes can not be released to infect the realistic users.

How to clearly evaluate a defense system is still a dilemma for security researchers. Several testbeds have been constructed to facilitate research into anti-worms.

2.3.1 vGround

vGround [22] is a virtualized environment to investigate worms' behaviors. The designing intention is to observe, record and analyze the exploitation patterns of worms. Virtual machines running real-world operating systems and application programs can be created and torn down easily. Virtual end nodes, switches and routers form the virtual networks, where the malicious worm codes will be released and propagate. The playground provides a conventional way to identify the probing behavior of worms.

The technique of User-Mode Linux (UML) [10] is chosen to support the virtual environment, which may contain several hundreds of virtual hosts in a single physical machine. Link-layer network virtualization makes the playground as an isolated sandbox and prevents the worms from leaking to the outside world.

However, the only network traffic in the testbed is worms' propagation.

The background traffic not simulated or replayed in vGround, since the concern focuses on the malicious packets containing worms' payload. The lack of background data makes vGround not suitable to evaluate new defense mechanisms.

2.3.2 DETER

The DETER testbed is designed for medium-scale repeatable experiments in computer security [15].

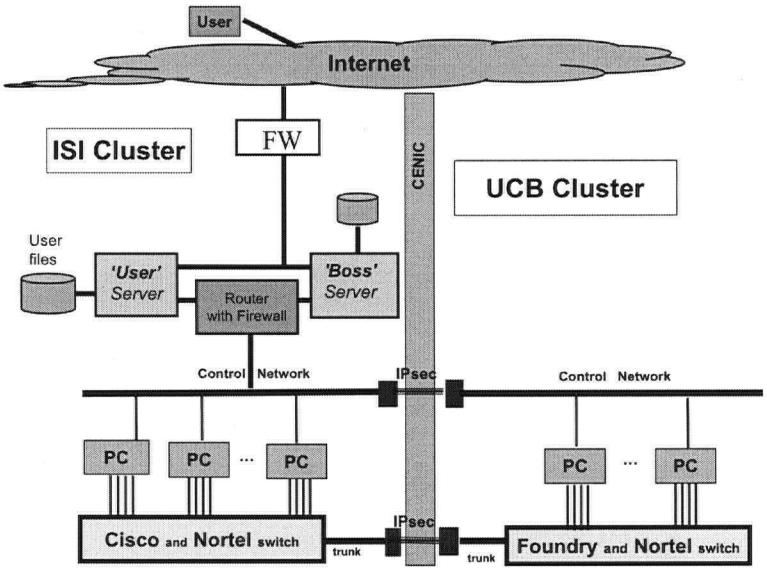


Figure 2.1: Scheme of DETER testbed [15]

The testbed is implemented in Emulab [2], where users have root privilege on a cluster of physical machines. As shown in Figure 2.1, OS images and file systems can be easily loaded from a 'User' Server. The test nodes are connected by switches to construct the experimental networks. The

deployment of real-world machines to evaluate defense systems offers high fidelity to security researchers. In order to enable remote access and confine the malicious codes within the testbed environment, several guard methods are deployed in DETER, including placing firewalls and intrusion detection systems on the outgoing path to monitor and filter the traffic. The purpose of DETER is to provide a flexible and safe experiment environment to test the defense mechanisms against the dangerous malwares.

DETER is a general prototype for security experiments. It does not specify the detailed implementation such as background replay or allocation of defense systems. Without virtualization techniques, the physical hosts will be compromised by worms, which aggravates the security problems. The control network is a weak point to be potentially attacked by worms. The experiments also suffer from the hardware restrictions, such as the number of physical network devices are fixed or the switches are not programmable by users.

2.3.3 Flexlab

Flexlab [28] tries to combine the advantages of PlanetLab (with real network conditions) and Emulab (with root privilege on the machines).

As shown in Figure 2.2, the rough idea is to monitor the traffic in PlanetLab (indicated as network model part), and then setup the corresponding application services in Emulab to replay the traffic with the same statistical characteristics, such as packet loss and latency. Meanwhile, Emulab grants complete control over the experiment.

However, Flexlab does not support to run malware, i.e. the emulation

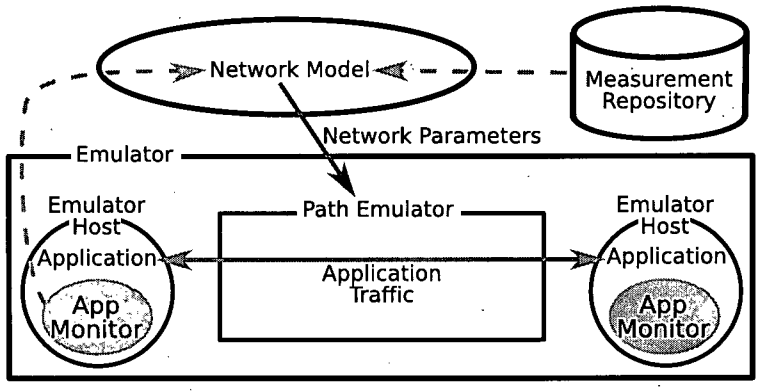


Figure 2.2: Scheme of Flexlab testbed [28]

of worm propagation is not embedded in the system. Therefore, Flexlab is insufficient for security experiments.

Anti-worm researchers look forward to an evaluation environment with realistic network conditions and malware propagation. Our testbed is designed to fill up the requirement.

Chapter 3

A Testbed to Validate New Enforcement Architectures

In this chapter, we present the basic structure of our testbed with the technology of virtualization. The overall goal of the evaluation system is to validate the enforcement architectures. An example of a generic testbed is shown in Figure 1.1, where the defense systems make judgements about whether the traffic in the network is normal or a part of a worms' attack. The detecting accuracy and the throttling impact are good measures of the defense ability of the anti-worm system.

3.1 Enforcing Fine-grained Security Policies

The conventional firewalls are widely used to protect an enterprise network. They focus on providing perimeter defenses and the internal local network is isolated from the outside internet. It is already hard enough for the network administrators to set firewall rules, make proper configuration and update software. Moreover, worms might be introduced by mobile devices, such as laptops, into the local network. In case that a host is infected, the firewall is not effective to protect the other machines from the worm attacks.

A technique used to overcome this problem is to watch all the individual machines [36]. Our assumption [12] is that an infected machine will try to connect to different machines as fast as possible. But an uninfected machine has a different behavior: the connections are made at a lower rate and are locally correlated. Monitoring on the network can detect the abnormal traffic.

The majority of the communication in enterprise networks uses the server-client model. The server side usually uses the fixed port numbers (a.k.a. well-known ports) for the clients to initiate the connections. In [25], the coupling information of servers, clients and ports are extracted by cluster methods. If a host suddenly sends many requests to the addresses and ports which scarcely appear in its connection history, an alert will be set on that host to indicate a suspect. But some application programs will use temporarily constructed ports for communication (called ephemeral ports). For example, FTP is a service in that it utilizes two ports, a 'data' port and a 'command' port (a.k.a the control port) [3]. Traditionally, on the server side the command port is 21 and data port is 20. But when communicating in the passive mode, the server sends its data port to the client and the client side will initiate the connection. The data port is specified in the command channel rather than being fixed in advance. The security mechanism in [25] currently could not correlate the traffic using ephemeral ports with the corresponding command traffic. If a fine-grained security policy is developed to track the states of the connections using ephemeral ports, it has the potential to greatly improve the detecting accuracy.

In order to monitor individual hosts and implement the fine-grained

policies, we need to attach the defense systems close to each end host to inspect the outgoing illegitimate traffic. The underlying hardware devices are assumed to have sufficient processing ability to analyze the packets (in layer 3) and are cheap enough to be widely deployed in a local network. Routers have great processing ability, but they are usually configured only at the main passageway of a local network and it is impratical to connect each machine with a single router. Switches are already universally deployed in the network to link machines or subnets, but most of them deal with layer 2 data only. It is natural to develop a kind of smart switch with the ability to process the packets at layers 3 & 4. Accordingly our testbed is required to easily support these layer 3 smart switches which embed the worm defense systems.

3.2 Virtualization Implementation

Due to the problems stated in Section 1.2, it is hard to construct a security testbed with physical machines. System virtual machines, i.e., virtualizing software emulating hardware abstractions [29], is a natural choice to support such a testbed.

3.2.1 Virtual Machines

A virtual machine (VM) is implemented by adding a layer of software to a real machine to support the desired virtual machine's architecture[29]. It provides users the illusion that an operating system or application runs on the real hardware. A typical structure of a virtual machine is as shown in

Figure 3.1.

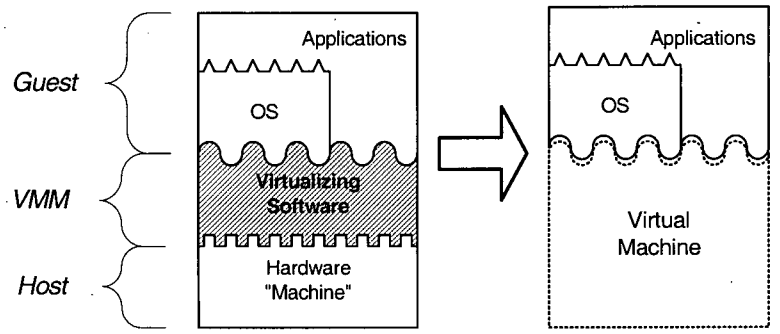


Figure 3.1: A system virtual machine [29]

The underlying platform is called the host, which provides hardware interfaces to the upper layers of software. The virtualizing software, usually referred to as the virtual machine monitor (VMM) or hypervisor, is set between the physical hardware and the conventional system software. On top of the VMM resides the guest operating systems and processes, which provide the desired functionality to the users. The VMM fills the gap between the hardware platform and guest system software, as shown in the right part of Figure 3.1, so the guest systems are unaware of the emulated virtualizing software and execute as in the actual hardware.

Multiple guest operating systems can run simultaneously on one physical host and share the underlying hardware resources. The VMM emulates many kinds of resources, including cpu, memory storage, networking, I/O, etc., to support the guest systems. Some of the emulated resources can not exceed the limit of the physical hardware, such as the total available cpu's cycles or the memory size; while other virtual devices won't be affected

by such restrictions, e.g., we are able to attach many NICs in the virtual machine. This flexible resource allocation allows us to scale the virtual machines to fit our experiment. With our concerns for security evaluation, we could create different versions of Windows, Linux or Mac systems with worm-exploitable vulnerabilities, and load the needed images into the virtual machines for evaluation.

3.2.2 Virtual Network

Since our research target is worm spreading, a virtual network environment, including cables, end hosts and routers, is required to support the experiment. Due to efficiency and security considerations, we implement the data-link layer in the virtualization network, which is similar to that in [22]. An example network configuration is demonstrated in Figure 3.2.

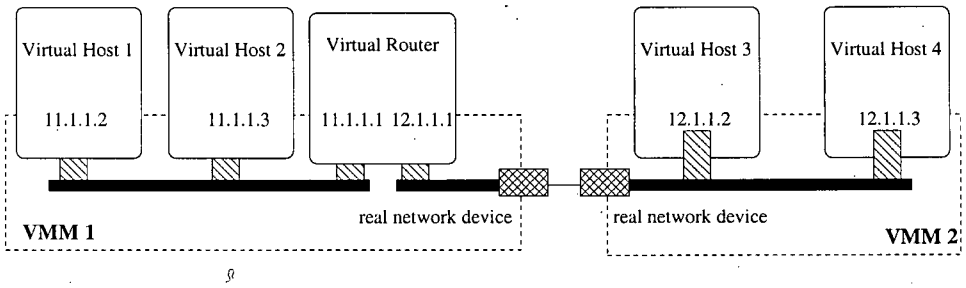


Figure 3.2: A virtualized network

The blank boxes are virtual machines, which are located in different physical machines. The virtual network will allow the virtual hosts, such as Virtual Host 1 (VH1), Virtual Host 2 (VH2) and Virtual Host 3 (VH3) to communicate with each other. Each virtual machine has one or several virtual NICs, indicated by the striped boxes. First, we demonstrate how

to connect the virtual machines hosted in one physical machine. The link-layer virtualization is implemented by attaching the virtual network devices to the linux bridges, which are indicated by the black boxes. The linux bridges are layer 2 devices which will forward all received frames to all but the incoming port. They function as real-world cables to link the network devices together, so the Ethernet data can be transferred from one end to the other. As shown in Figure 3.2, a data-link path is then constructed from VH1 and VH2 to the virtual router, so all virtual machines with IP addresses in the same subnet can access each other. During the forwarding process, packets will not traverse up to the IP layer, which is more efficient as resolving the headers of advanced protocols is not necessary. Although it is possible to support hundreds of guest systems in one single physical machine, the virtual machines are sometimes located in different physical hosts due to resource limitations or special requirements. In order to communicate between virtual hosts in different physical machines, the hardware network devices (shown as grid boxes) connect the internal virtual network and the outside actual-world cable. Since no IP addresses are assigned to the physical network devices, the data-link layer virtualization is maintained. By this means, the virtual machines in the same subnet but on different physical machines can communicate with each other. If there are more than one network segment in the experiment, we need to bring up virtual routers, and they will function as the real-world routers to translate the packets at layer 3. Then the virtual hosts in different physical machines and in different subnets respectively are able to talk to each other.

The simulated network allows the virtual hosts to send packets through

the TCP-IP stack, traverses the data through the intermediate network and eventually makes the packets arrive at the destination. The virtual hosts are unaware of the underlying software-implemented connections. We can manipulate the MAC addresses, IP addresses and routing tables of the end hosts, deploy various routing algorithms on the virtual routers, and thereby configure whatever network topology is necessary to facilitate a good experiment.

3.3 Testbed Structure

Currently we assume that the testbed is a closed environment, i.e., there are no network connections to the outside world and users are required to be physically at the testbed to use it. We will remove this restriction in Section 4.3. Our implementation is illustrated by the small-scale example in Figure 3.3. The worm experiments actually run in the virtual hosts and the virtual networks, and are supposed to be contained in our test environment.

There are 3 physical machines in our example. The dotted boxes represent physical hosts. The solid rectangles with 'VM' are the virtual machines supported above the hardware. The black boxes are the standard linux bridges which connect the virtual hosts (including the virtual switches and virtual routers). They not only act as virtual cables for network connections, but play an important role for replaying the background traffic as well. That is the reason why each linux bridge is only attached to 2 virtual network devices in our architecture. The virtual switches (as shaded boxes in Figure 3.3) are not prerequisite to allow the packets going through the

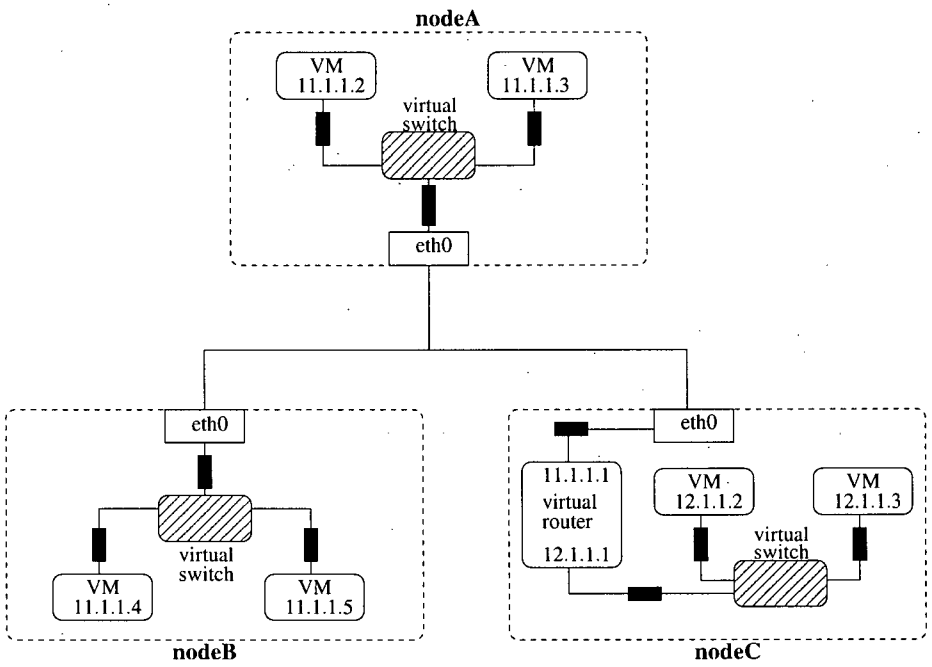


Figure 3.3: Topology of the worm experiment

network, since linux bridges are enough to glue all the virtual hosts (including the virtual router) together. We build up this kind of special virtual machines for the embeded defense system to filter the traffic in the virtual environment.

In Figure 3.3, we assign 2 network segments to construct the virtual networks, 11.1.1.* and 12.1.1.*. There is a virtual router in nodeC to connect those 2 virtual subnetworks. Due to the virtualized implementation, the virtual hosts could be placed on any physical machines, as long as we configure the routing tables and the underlying bridge connections correctly. Each virtual host attaches to a bridge which leads to a virtual switch. Since the bridges will record the MAC addresses on their sides and forward the

packets to the corresponding ports, an individual virtual host will only receive the packets sent to its address, and is unable to eavesdrop on traffic destined to others. Once the worm codes are put in some of the virtual hosts, they will begin to propagate on their own in the virtual network and infect the vulnerable systems. The dumped background traffic is injected from the linux bridges. Then the defense mechanisms running in the virtual switches will take care to monitor and filter the traffic

3.4 Deployment of Defense Systems

All our effort is constructing a convenient and realistic environment to evaluate the defense ability of an anti-worm system. Unlike the conventional firewall-based defense systems, the switch-based mechanism (as stated in Section 3.1) aims to isolate each end host from the rest of the network. The rough idea is that once an individual machine is convicted of sending abnormal traffic, the packets from that machine will be throttled or even blocked. In order to develop such a defense mechanism, we need smart switches placed between each corresponding end host and the rest of the network.

Since smart switches are the key elements of our defense technology , we need to decide how to implement them in the evaluation testbed. The virtual switches (the shadowed boxes in Figure 3.3) are implemented with full virtual machines, which run the typical operating systems and application programs. Multiple virtual network cards are assigned in such switches, each of which connects a virtual end host via the linux logical bridge. In order to maintain the data-link layer virtualized network, a virtual wire is con-

structed with the virtual switches, i.e., a bridge is generated in each virtual switch to connect its virtual NICs as well. Then the packets will traverse the switches and be forwarded to the destination.

The virtual smart switches have the following capabilities, which makes them fit the purpose of security evaluation.

- Since the virtual switches are standard virtual machines, they provide a flexible platform to install the user-specified defense systems. Many application programs and libraries are available for users to filter the packets. The straightforward method is to setup firewall rules to filter the traffic, and researchers are encouraged to develop their own defense algorithms;

- When listening on the bridge within a switch, all the passing traffic from the end hosts linked to that switch can be captured. The switches are able to monitor the packets and run arbitrarily complicated algorithms to analyze the traffic;

- Because the number of network devices created in a virtual machine has no hard limitation (theoretically), we can configure the virtual network and connect each virtual switch to as many end hosts as necessary . Users are thus able to collect the packets of several interested end hosts in a single virtual switch and do the judgement based on the overall traffic, as long as those virtual hosts are connected to the same switch.

We assume that the anti-worm system on different 'smart' switches will

not communicate with each other to detect and throttle the worms. Such distributed defense system is not the focus of our work.

Chapter 4

Design Details

In this chapter, we further introduce the features of the virtual testbed and explain the design details, including the methods to capture the packets, to replay the background traffic, to prevent worm leakage and to allocate the resource for the domains, etc. All the design considerations are mainly based on the virtual infrastructure.

4.1 Key Technologies

4.1.1 Virtualization (Xen)

A number of virtual machine systems have been developed, which are able to virtualize the hardware, so that several operating systems can share it. Such software products include VMware [11], Denali [34], Xen [14] and User-Mode Linux (UML) [10]. We choose Xen to support our testbed, however the ideas can be applied by using any other virtualization technology.

The reasons for utilizing Xen include 1) Compared with other virtualization techniques, Xen achieves high performance on the x86 architecture. The performance of guest systems over Xen is practically equivalent to the performance of the baseline linux [14]; 2) Xen's motivation is to run a mod-

erate number of full-featured operating systems, which fits our design to emulate a large network with various operating systems and implement the programmable 'smart' switches. 3) The underlying Xen hypervisor allows several options to manage the network connections. One of them is to run the networks with logical ethernet bridges. Such a configuration is critical for us to implement the data-link layer connections and replay the background traffic; 4) Unmodified guest operating systems are enabled to run within Xen virtual machines, starting with Xen 3.0. Therefore without making any changes in the guest systems, people can perform tests on the worms exploiting vulnerabilities in different Unix-like systems and Microsoft Windows.

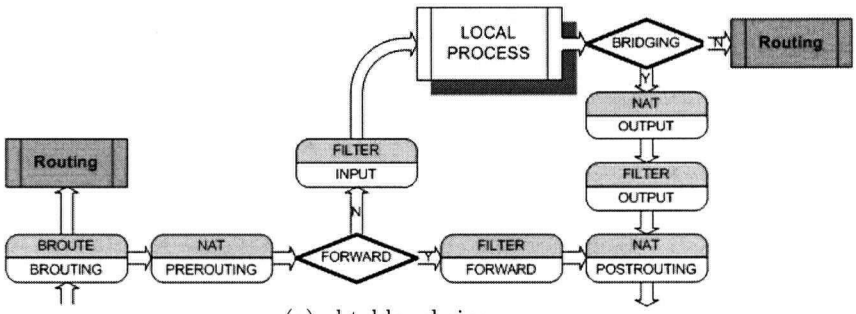
In Xen, the term guest operating system refers to one of the OSes that Xen can host and we use the term domain to refer to a running virtual machine within which a guest OS executes [14]. A special domain, named domain0, is created in Xen to control and manage the other domains. In domain0, we can configure, create, terminate, and monitor the virtual machines, and specify what resource are allocated to each domain.

4.1.2 Iptables & Ebtables

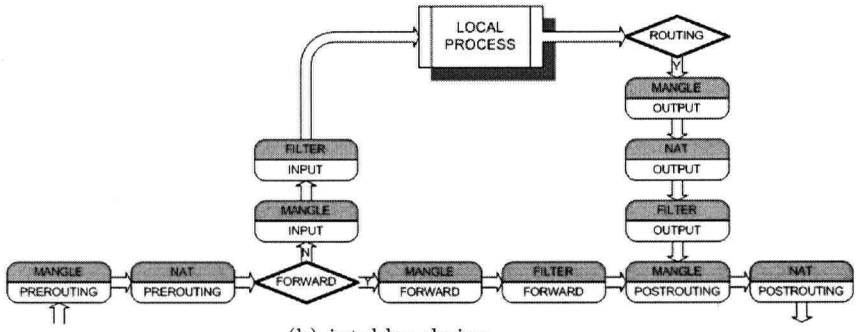
Linux systems provide a set of hooks within the kernel for intercepting and manipulating network packets. The filter framework at the IP layer is Iptables [4]; while that at the Ethernet layer is Ebtables [1]. We can utilize either technology to implement the candidate defense algorithms and control the virtual network.

Ebtables works on the logical bridges. Kernel module codes, called

chains, are attached to the different hooks in the bridge to process the packets, including BROUTING, PREROUTING, INPUT, FORWARD, OUTPUT and POSTROUTING chains, as shown in Figure 4.1(a).



(a) ebtables chains



(b) iptables chains

Figure 4.1: Traversal scheme of network filters [1]

The packets will go through different chains based on their destination MAC addresses. In our testbed, we focus on the INPUT, FORWARD and OUTPUT chains. After the incoming frames pass the BROUTING and PREROUTING chains, if the bridge decides the frame is destined for the local computer, the frame will go through the INPUT chain. Attaching to the INPUT chains allows us to filter the frames destined for the bridge before they are passed up to the network layer. Otherwise, the bridge will forward

the frames to other machines and make them go through the FORWARD chain, where we can filter the bridged frames. Locally created frames will, after the bridging decision, traverse the OUTPUT chain, which allows us to filter the frames originating from the bridge box.

Iptables filters packets at the network layer. It has the similar chains to process the packets, as shown in Figure 4.1(b). Our focuses are still the three basic chains (INPUT, OUTPUT, and FORWARD), and the users can specify the filter rules based on the source or destination addresses, application-layer protocols and working network devices, etc. One of the important features built on top of Iptables is connection tracking. It allows the kernel to keep track of all logical network connections or sessions. The network connections with protocols using ephemeral ports could be traced by the defense systems to implement the fine-grained security policies stated in Section 3.1. The functionalities of Iptables and Ebtables are convenient for users to configure their own smart switches.

4.1.3 Implementation Infrastructure (Emulab)

In order to construct our testbed, several physical machines connected within a local network are required. Emulab [35] provides such a environment: fully-privileged machine nodes and an easily-configured network. There are other system and network emulators, such as Modelnet (which emphasizes scalability) [32] and PlanetLab (which emphasizes services) [5], but we choose to use Emulab for the following reasons.

1) Emulab provides realistic machines for users. Users are granted root privilege to install software, manipulate kernels and configure the hardware.

The detailed hardware setting is described in [2]. It is noted that each node has 5 Intel EtherExpress Pro GigE Ethernet network interface cards. Four of them can be configured by users to form experimental networks, the remaining one is reserved for remote control by Emulab.

2) Emulab allows users to create their own custom OS images. Users can then specify to load them into the nodes automatically when an experiment is created. It is convenient for us to configure the physical machines running Xen and to adjust the scale of the testbed.

3) Users specify the NS scripts to configure the experimental networks in Emulab. High-speed Nortel switches are deployed to connect the end nodes automatically.

4.2 Background Traffic Replay

Realistic background traffic is critical to accurately evaluate the usability vs. security of an anti-worm mechanism, since the underlying intuition of detection is to find out a good feature to distinguish normal traffic from malicious traffic. We will first capture the traffic in a real-world local area network (LAN) and then inject these network data into the testbed.

4.2.1 Data Collection

The pcap (Packet Capture) library provides an efficient way to capture the complete packet data transferred on the wire. The Linux version of pcap, libpcap [6], utilizes the BSD Packet Filter (BPF) [24] to receive and send raw link-layer packets and filter the packets at the kernel level. That utility

fits our purpose of replaying the background traffic.

The packets dumped via pcap are absolute copies of the link-layer data from the network devices, e.g., a TCP packet including the Ethernet header, IP hearder, TCP header and the corresponding application data. Moreover, pcap will encapsulate the packet with its own header, where some important information, such as the packet capturing time, is recorded. To collect the background traffic, we will listen on an enterprise LAN to dump the passing packets. In order to map the topology of a real network into the virtual environment, it is required to know the list of host addresses and how they are connected by switches and routers. However, due to the privacy consideration, the IP and MAC addresses in the enterprisze network might be replaced by artificial ones, and the content at the application-layer will be changed to random bytes while being kept the same length.

Besides the binary format, another way of storing the traffic trace is to record them in plain text. The required components include protocols, addresses, ports (if the packets are TCP or UDP) and time stamps. During the replaying process, the whole packets are reconstructed form the text data trace. Compared with tcpdump data, the traces in text format need more time in the user-level to preprocess the packets, so we prefer the data collection with tcpdump [6] in this thesis.

4.2.2 Packets Replay

The tcpdump format data can be directly sent out from the network devices to the cables. If in each virtual host we replay the corresponding packets with the same source IP address as that of the host, the overall background

traffic will appear in the virtual network and be ready for being analyzed by the defense system under test.

However, the virtual hosts run various kinds of operating systems, so different programs would need to be developed for different systems. In order to avoid making changes in the virtual hosts, we move the replaying functionalities into the virtual machine monitor. In Figure 3.3, the packets are injected into the logical bridges close to the virtual hosts, instead of in the nodes themselves. A modified replaying program in Xen's hypervisor can handle all the replaying events. The whole data trace will be split based on the source IP addresses and each of the split dump files will be replayed on the corresponding bridge. For example, in Figure 4.2 the packets from VM A are sent out at the logical bridge which is connected to its domain. Layer 2 bridges forward the packets based on the destination MAC addresses. Then the replayed traffic will traverse the virtual network, go through the smart switches and arrive at the destination VM B.

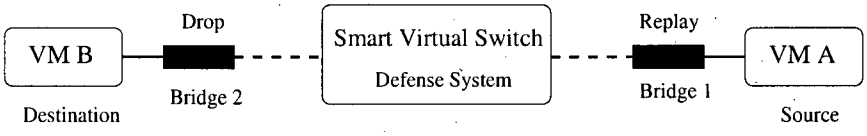


Figure 4.2: Replay scheme

If the replayed packets hit VM B, B might generate replies for some request packets. But the replied packets from VM B are in the dump file as well and will be replayed at bridge 2. Then the replayed packets from VM A can not be actually received by B. In order to solve that problem, we enable the Ebtables function to drop the replayed packets from VM A at the

bridge close to VM B. As the replayed packets will pass Ebtables' OUTPUT chain at the sending bridge, but go through the FORWARD chain at the receiving bridge, if we set the Ebtables rules as ACCEPT at OUTPUT chain and DROP at FORWARD chain for each bridge, all the packets to the end hosts will be dropped at the bridges.

It is noticed that the live traffic between virtual hosts are blocked by the Ebtables rules as well, which disables the real worms spreading in the testbed. So we need to distinguish between the replayed traffic and live traffic. The solution is to change some unused bits in the dumped packets' headers to a specified value. Once Ebtables detects such marked bits in the packets, they will not be forwarded at the bridge. Therefore, the replayed packets can not reach the end hosts, but the live traffic between the virtual machines will come through the network without problem.

4.2.3 Synchronization for Causality

The replayed packets will be analyzed by the defense systems, so it is important to keep the packet causality of the connection. The sequence of the packets should be messed by the replaying process, e.g., the request packets in a connection should not appear in the testbed before the corresponding reply packets. In order to synchronize the packets, we monitor the traffic on the bridges during replaying.

As shown in Figure 4.3, we demonstrate the communication between a host-pair A and B, where the packets from A are labelled with English characters, and those from B are indicated by numbers. The first packet x is read in at both bridges. At the side of A, the replaying process finds out

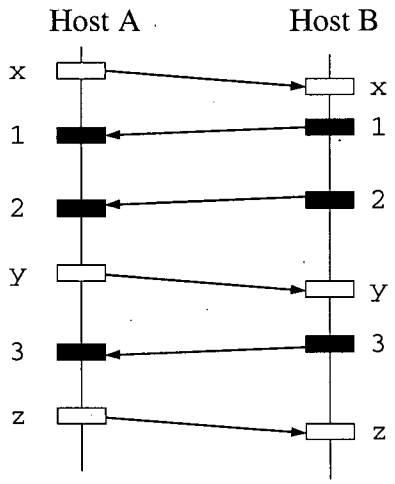


Figure 4.3: Synchronization for causality

x originated from A, so the packet is directly sent out to the network. The bridge at B analyzes the received packet and figures out that x is destined to B. Then the program will listen on the bridge to wait for the incoming packet x from A. Once the bridge at B receives x, it will continue to process the next packet. After the recorded time interval, packet 1 will be sent at the bridge close to B. Meanwhile, the bridge at A is waiting for packet 1 to appear on the network. This issue-and-wait process will repeat in turn at the bridges of both sides. Therefore the traffic trace '$x, 1, 2, y, 3, z$' is correctly replayed in the virtual network.

4.3 Isolated Details

Now that the actual worms are released in our testbed, the simplest way to guarantee no leakage of malicious codes to the outside real world is to make

the testbed isolated from other networks and run the experiments at the desk. However, users hope to remotely access and control the experiments. In this section, we investigate the methods to strictly confine the worms in the testbed with remote connection enabled.

4.3.1 Data-Link Layer Virtualization

Within our testbed the real worms reside in the virtual machines and try to infect the other vulnerable hosts. In case that the physical machines running Xen were accessible by network connection from the virtual machines, they could be attacked or even compromised as well. The design of the virtual network (in Section 3.2.2) makes sure that will not happen.

The virtual networks are implemented at the Ethernet layer. The transport network devices, including logical bridges, smart virtual switches and physical NICs for the experimental networks, do not have IP addresses, so they will forward the frames only based on the MAC addresses.

So far no MAC-based worms have been discovered, i.e., the worms will scan the IP address space and exploit the vulnerability of the application services. Therefore the above mentioned physical and virtual devices will not forward the packets up to the application layer. Especially, the scanning worms are not aware the existence of the underlying physical machines nor are they able to establish network connections with them. The worms, which launch network attacks, are unable to infect the supporting physical machines in the testbed.

4.3.2 VLAN Technique

There are many different experiments running in Emulab, and it will cause disaster if the worm traffic in our testbed could reach the physical machines in other experiments.

The technique of VLAN (virtual LAN) alleviates such worries. The underlying networks in Emulab are constructed by physical (Nortel) switches attached with end physical hosts. The machines in one experiment will behave as if connected to the same link layer network, since the physical switches will forward packets only to those ports specified in the experimental VLAN. Therefore, the traffic in one experiment's VLAN will not appear in others. Even if the testbed are located on more than one physical machine, where the malicious traffic goes through the real-world wires, it is impossible for the worms to reach the physical machines in other experiments.

4.3.3 Remote Control

The remote control is a potential path for worms to leak to the outside world. Emulab deploys the firewalls to isolate the testbed environment from the outside network. However, in order to block various kinds of malicious worms, the firewalls have to be updated with the worms' signatures. Our design fundamentally solves the security problem of enabling remote access to the testbed.

In Emulab, each physical machine has a separate NIC for remote control. We take advantage of this architecture. Our configuration is that the NIC connected to the experimental network is not assigned any IP address; while

the NIC for remote control will have its IP address. Moreover, the control NIC does not attach with any other network devices in Xen. Therefore the worms have to bypass the physical machines in order to attack the remote user via the control NIC. As long as the physical machines are not compromised (stated in Section 4.3.1), the remote control network is secure. The complete structure of our testbed is shown in Figure 4.4, where the network devices of *eth2* are for the control network.

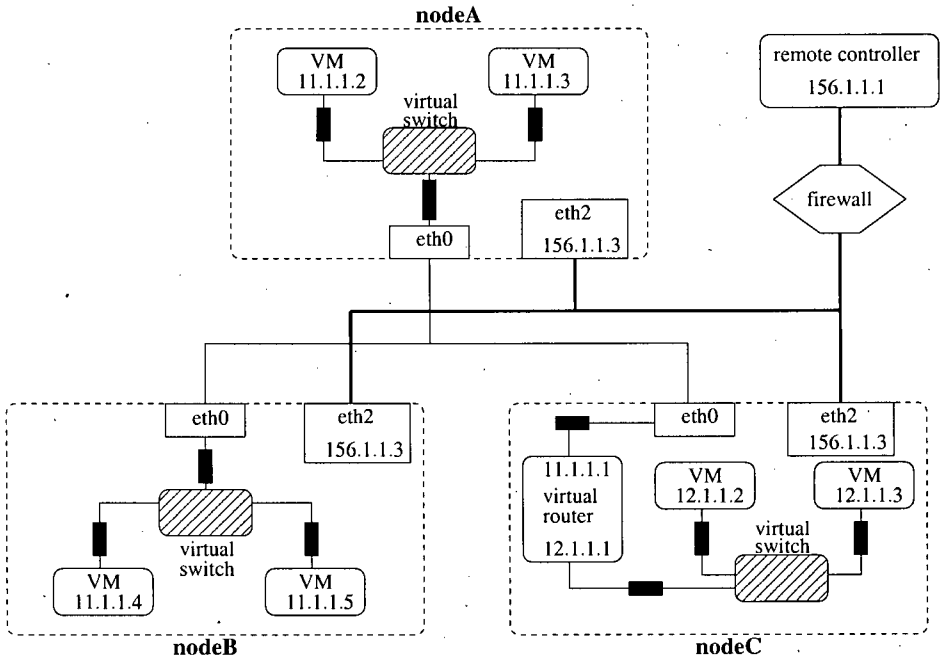


Figure 4.4: Structure of the worm experiment with remote control

4.4 Resource Allocation

Since tens or hundreds of virtual hosts are supported in each physical machine, we need to reasonably allocate the resources for the domains to make the testbed run normally.

4.4.1 Time Dilation

We execute an user-level process for replaying the traffic between each pair of hosts. Suppose the number of end hosts is N, then there are $O(N^2)$ replaying processes. If the scale of the experiment is too large and the available physical machines are not enough, the traffic will not be replayed accurately and some packets will be lost. In order to reduce the CPU burden for processing the packets, we slow down the traffic replaying speed in domain0. Correspondingly, the time passage in the end hosts and the smart switches should be slowed down with the same factor as well. So we enable the time dilation [20] in Xen's guest domains. After being specified a time dilation factor (TDF), the ticks in the domains will become TDF times that in the real world. As we set the TDFs of replaying processes and in guest domains as the same, the guest domains (including the virtual end hosts and the virtual switches) feel no difference about the network events no matter whether the time dilation is enabled or not.

4.4.2 Parameter Setting

We assume that users will construct one smart virtual smart switch on each physical host. Suppose there are a total of K physical machines involved in

the testbed, n_k is the number of virtual machines in physical host k and ki indicates the ith virtual host in it ($i < n_k$). Within the physical machine k, the required CPU resource for each virtual end host is h_{ki}; the CPU resource for the smart virtual switch is s_k; and the CPU resource for Xen (domain0) is x_k. Further, assume the total available CPU power on machine k is c_k. Usually, $\sum_{i=1}^{n_k} h_{ki} + s_k + x_k > c_k$, i.e., the required resource is greater than the available resource. If we set the time dilation factor in machine k as T_k in the below formula, the domains in machine k will have enough CPU resource.

$$T_k = \frac{\sum_{i=1}^{n_k} h_{ki} + s_k + x_k}{c_k}$$

In order to make the time passage on different physical machines consistent, we set the time dilation factor of the testbed as the maximum ratio for all physical machines, $T = \max_k T_k$. Such setting will ensure that each domain has sufficient CPU resources.

Chapter 5

Evaluation

In order to evaluate the effectiveness of our testbed, we test the accuracy of the replaying procedure. The dumped real-world traffic reoccurs in the virtual network. Specifically, the experiments are performed with two main goals: to verify 1) replaying procedure maintains the packets' statistics. 2) the time dilation will not impact the fidelity of the background traffic. We first describe the experimental setting in Section 5.1. The evaluation results are then presented in Section 5.2.

5.1 Experiment Setting

Since the raw packets on the network reveal the topology information and the application data, few enterprises or organizations are willing to share their LAN's internal traffic due to the above mentioned privacy concerns. Fortunately, the appearance of the DARPA data set alleviates the problem of lacking data for security experiments.

The 1999 DARPA Intrusion Detection Evaluation Program [23] was prepared and managed by MIT Lincoln Labs. The purpose was to evaluate research in intrusion detection. Lincoln Labs set up an environment to acquire nine weeks of raw TCP dump data for a local-area network (LAN)

simulating a typical U.S. Air Force LAN. As show in Figure 5.1, hundreds of different types of users, including programmers, workers, managers and system administrators, etc., were emulated in the LAN. The network traffic involved over 20 kinds of services, including dns, finger, ftp, http, ping, pop, snmp and telnet. Therefore the characteristics of the automatically created traffic represent those of realistic traffic. Especially, the application services, such as email, ftp and web, are close to the real-world communication. The contents of the packets are mainly collected by 2 ways: one is from public documents, and the other is from syntax statistics.

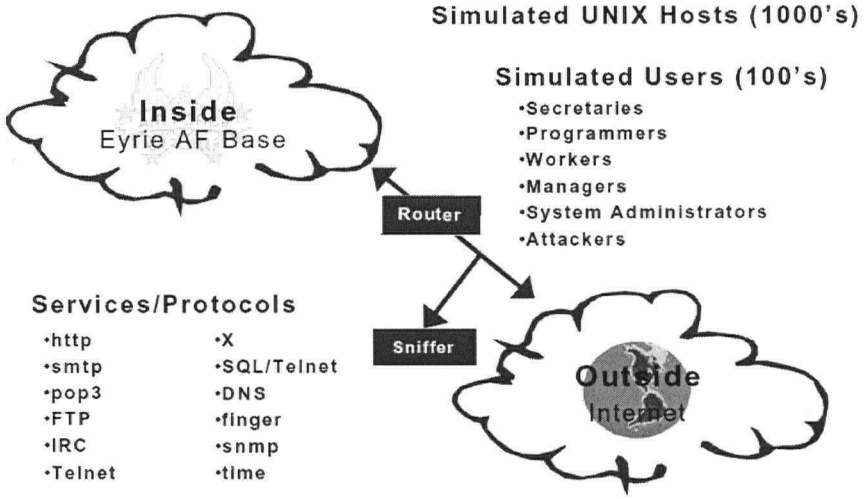


Figure 5.1: DARPA simulated network [9]

Figure 5.2 shows the network structure of the 1999 DARPA intrusion detection evaluation program. The networks are partitioned as 2 parts: inside and outside. The air force LAN is emulated in the inside network, where 4 machines are set up as attacking targets with OS of Linux 2.0.27, SunOS

4.1.4, Sun Solaris 2.5.1 and Windows NT 4.0 respectively. Meanwhile, a gateway is deployed to emulate hundreds of internal machines. In order to simulate the outside traffic, one machine emulates the traffic of the outside network; another machine is used to simulate web services. Although one significant feature of DARPAevaluation program is that part of the data contain various typical network attacks, for the purpose of validating the fidelity of our replay mechanism it is sufficient to run the experiment on the attack free data set.

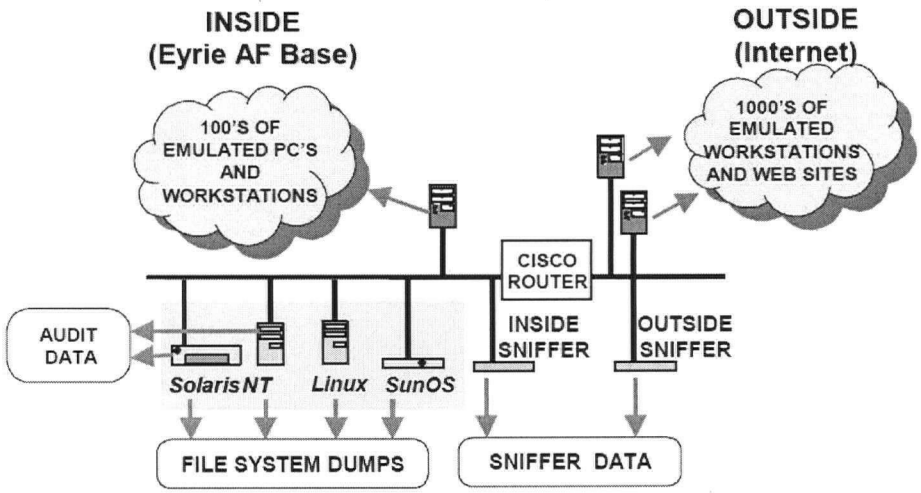


Figure 5.2: DARPA network topology [9]

We show the average TCP connections during one day in Figure 5.3. Everyday, about 441M bytes of data were transferred on the simulated network. The running time is roughly from 8:00 am to 6:00 pm. The major protocols include TCP (384M bytes), UDP (26M bytes) and ICMP (98K bytes). However, our replaying focuses on the internal communication, so

we discarded the simulated outside traffic in DARPA data trace. Then the size of the dump file will reduce to 10% of its original size. Since our experiment only replayed the internal traffic in the DARPA data, we just reconstructed the 28 internal machines in our testbed, and didn't simulate the outside Internet (as shown in Figure 5.1).

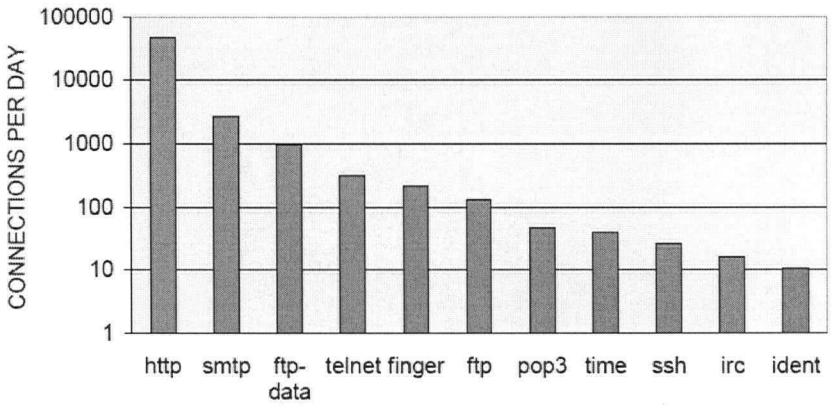


Figure 5.3: Statistics of daily TCP services [9]

Due to the function of gateways, the machines with different IP addresses do not have distinguished MAC addresses in the DARPA data trace. So we manually constructed a mapping between MAC and IP addresses, and changed the corresponding MAC addresses in the dump files. As stated in Section 4.2.2, we mark the source MAC addresses in the dump files to be ended with byte 0x22, while the individual virtual hosts' MAC addresses are ended with byte 0xdb. Therefore the Ebtables function on the intermediate virtual switches will not forward the replayed packets, but allow the live traffic through.

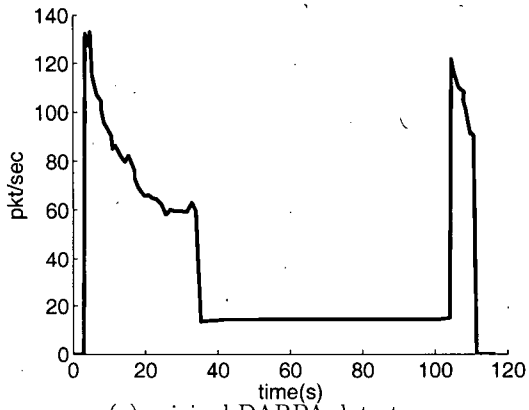
For simplicity, right now we run all the experiments on a single test

node in Emulab. The machine in UBC Emulab has Intel(R) Pentium(R) 4 CPU 3.20GHz and 512M bytes memory. We set the minimum memory size for domain0 to 196M bytes and the memory size of virtual smart switch to 128M bytes. The memory allocated for each virtual end host is 16M bytes, since the end hosts do not require too much resource for the replaying test.

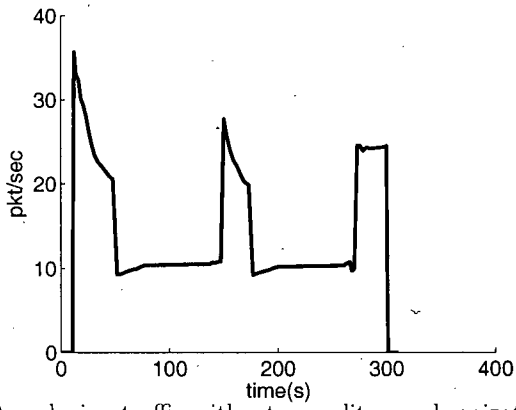
5.2 Experiment Results

We first focus on the traffic replaying between 2 machines. We chose the hosts with IP addresses of 172.16.112.20 and 172.16.112.100 in the DARPA data set as a communication pair. The connection of these 2 hosts is similar to the configuration in Figure 4.2. Host A and B are set with 2 IP addresses, and the dumped packets are replayed at bridge 1 and 2 respectively. We monitored the replayed packets in the virtual switch to compare the throughput with the original data trace.

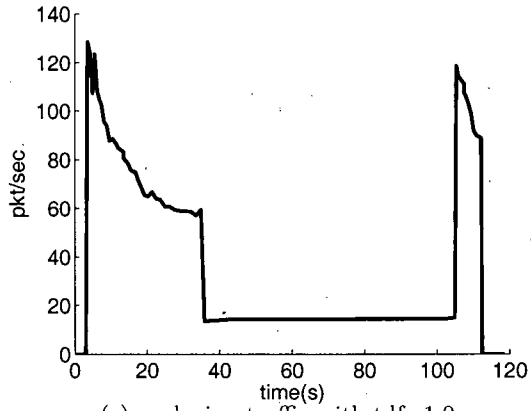
The throughput of the original DARPA data is shown in Figure 5.4(a), which has the peak of around 140 packets per second. If we simply replay the packets regardless of the causality, the throughput on the network will be quite different. It will take a long time to replay with a lower rate as shown in Figure 5.4(b). It is obvious that replaying time is greatly delayed, and the peak throughput (37 packets per second) is decreased. When we correctly maintain the packets' sequence by waiting the opponents' packets, the throughput (shown in Figure 5.4(c)) matches that in the DARPA data trace. As long as the processor has enough ability to proceed the dumped packets, the replayed traffic will have the similar statistical characteristics



(a) original DARPA data trace



(b) replaying traffic without causality synchronization



(c) replaying traffic with tdf=1.0

Figure 5.4: Replaying between 2 machines

as the original traffic.

Next we test the effect of time dilation on the replaying process. The replaying speed is slowed down by a factor of 10, and meanwhile the time passage in the virtual switch is slowed by the the same factor. The throughput measurement in the virtual switch is shown in Figure 5.5. The result is still consistent with that in DARPA data trace, so this verifies that from the point of view of the virtual machines, the replaying speed is not changed after time dilation.

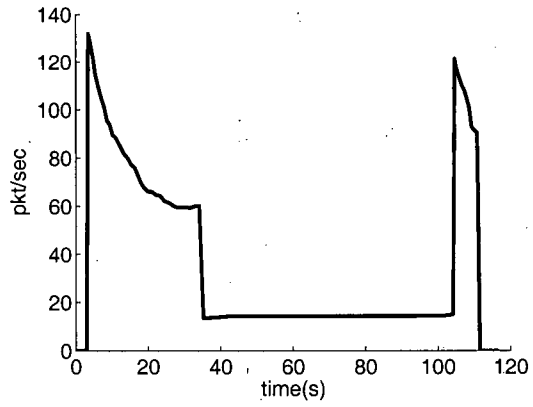
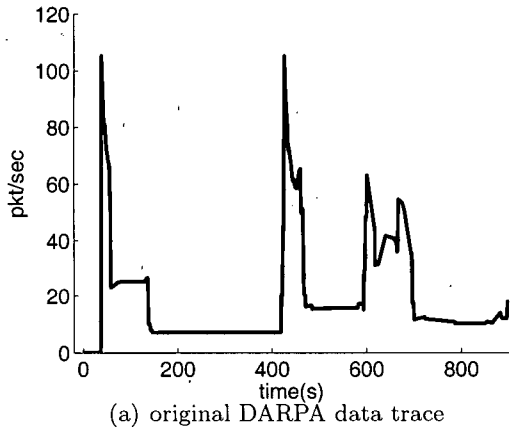


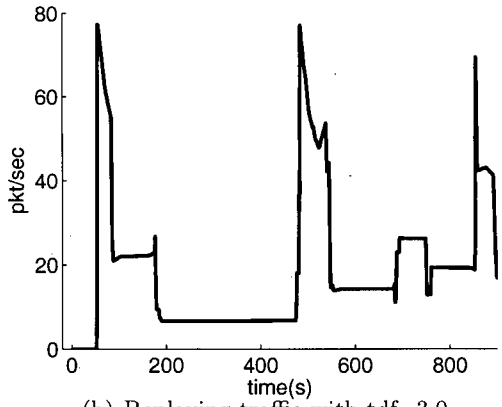
Figure 5.5: Replaying between 2 machines with tdf=10.0

We simulated the whole local network with 28 hosts specified in the DARPA 99 trace as well. All the virtual hosts are connected to the single virtual switch. A segment of the data trace is shown in Figure 5.6(a).

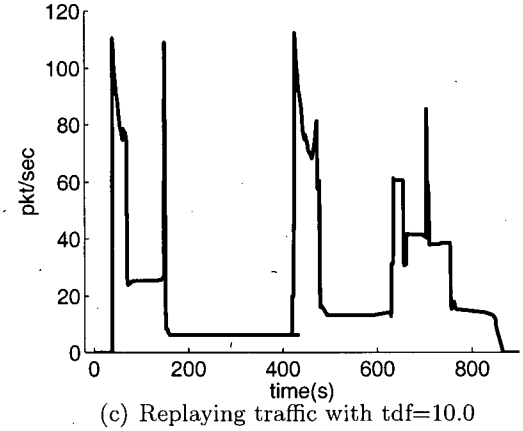
As the number of replaying processes have been dramatically increased compared to the previous experiment, the resources are not enough to execute the replaying processes. When we replayed the packets with a small TDF, e.g., set TDF as 3, the lack of resources would cause much delay as



(a) original DARPA data trace



(b) Replaying traffic with tdf=3.0



(c) Replaying traffic with tdf=10.0

Figure 5.6: Replaying in the whole DARPA network

shown in Figure 5.6(b), i.e the processing of replaying packets can not cope with the original traffic speed. When we increased the TDF to 10, the processing ability would handle the replaying speed. The result became better in Figure 5.6(b). The method of time dilation is effective to alleviate the resource limitation. However, a side effect is that the experiment will run TDF times longer.

Our experiments validate that the replaying mechanism will maintain the fidelity of the background traffic. Even with time dilation, the statistics of the traffic will appear the same from the view of the virtual machines.

Chapter 6

Conclusion & Future Work

In this thesis we proposed a virtual testbed to facilitate the evaluation of the usability vs. security of anti-worm algorithms. Our main contributions include (1) The virtualized implementation allows the scale of the experiment to be expanded or reduced without hard limitation, e.g., tens or hundreds of virtual machines can be set up in a single physical host; (2) We maintain the high fidelity of the background traffic by deliberately replaying the dumped packets; (3) Since the real-world operating systems and application programs run in the virtual machines, we can introduce the realistic worm codes into the virtual environment and mix them with background traffic for testing; (4) The behaviors of the layer 3 switches are accurately emulated, so researchers can evaluate switch-based security policies in our testbed; (5) The malicious traffic is strictly confined within the testbed.

Our work is still an ongoing project, and we plan to address several future directions. First, the mixture of live and replayed traffic may cause some problems. For example, in a TCP session, the port numbers may conflict in the two different types of traffic. If some defense systems trace the connection based on the port numbers, such conflicts will make the defense system become confused. Therefore, the replaying mechanism needs further

adjustment when being combined with worm propagation. Second, currently we just built up the testbed on a single physical machine. In principle the virtual machines can be located on several different physical hosts. The emerging problems are how to reasonably assign the virtual machines into the given physical machines and whether users can change the assignment during the experiment; Finally, in order to verify the effectiveness of our testbed, we need to deploy some defense systems in our testbed and compare their defense abilities. Basically, the future direction is to make the testbed more flexible and reliable to evaluate the worm defense systems.

Bibliography

[1] Ebtables. http://ebtables.sourceforge.net/.

[2] Emulab. http://www.emulab.net/.

[3] Ftp rfc. http://rfc.net/rfc959.html.

[4] Iptables. http://www.netfilter.org/.

[5] Planetlab. http://www.planet-lab.org/.

[6] Tcpdump. http://www.tcpdump.org/.

[7] CERT. CERT advisory ca-2001-26 nimda worm. http://www.cert.org/advisories/ca-2001-26.html.

[8] eEye Digital Security. http://www.eeye.com/html/research/advisories/al20010717.html.

[9] MIT Lincoln Labs, DAPAR intrusion detection evaluation. http://www.ll.mit.edu/IST/ideval/.

[10] User Mode Linux. http://user-mode-linux.sourceforge.net/.

[11] Vmware. http://www.vmware.com/.

[12] William Aiello, Charles Kalmanek, Patrick McDaniel, Subhabrata Sen, Oliver Spatscheck, and Jacobus Van der Merwe. Analysis of communities of interest in data networks. In *PAM'05: Proceedings of 6th Passive and Active Measurement Workshop*, 2005.

[13] Julia Allen, Alan Christie, William Fithen, John McHugh, Jed Pickel, and Ed Stoner. State of the practice of intrusion detection technologies. *Carnegie Mellon University, Technical Report CMU/SEI-99-TR-028*, 1999.

[14] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP'03: Proceedings of 19th ACM Symposium on Operating Systems Principles*, 2003.

[15] Terry Benzel, Robert Braden, Dongho Kim, Cliford Neuman, Anthony Joseph, and Keith Sklower. Experience with deter: A testbed for security research. In *TRIDENTCOM'06: Proceedings of 2nd IEEE Conference on testbeds and Research Infrastructures for the Development of Networks and Communities*, 2006.

[16] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX'05: Proceedings of 14th USENIX Security Symposium*, 2005.

[17] David Brumley, Li-Hao Liu, Pongsin Poosankam, and Dawn Song. Taxonomy and effectiveness ofworm defense strategies. *Carnegie Mellon University, Technical Report CMU-CS-05-156*, 2005.

[18] Zesheng Chen, Lixin Gao, and Kevin Kwiat. Modeling the spread of active worms. In *INFOCOM'03: Proceedings of 26th Annual IEEE Conference on Computer Communications*, 2003.

[19] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *HotOS'97: Proceedings of 6th workshop on Hot Topics in Operating Systems*, 1997.

[20] Diwaker Gupta, Kenneth Yocum, and Marvin McNett. To infinity and beyond: Time-warped netowrk emulation. In *NSDI'06: Proceedings of 3rd Symposium on Networked Systems Design and Implementation*, 2006.

[21] Herbert W. Hethcote. The mathematics of infectious diseases. *Society for Industrial and Applied Mathematics*, 42(4):599–653, 2000.

[22] Xuxian Jiang, Dongyan Xu, Helen J. Wang, and Eugene H. Spafford. Virtual playgrounds for worm behavior investigation. In *RAID'05: Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection*, 2005.

[23] Matthew V. Mahoney and Philip K. Chan. An analysis of the 1999 darpa/lincoln laboratory evaluation. In *RAID'03: Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection*, 2003.

[24] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX'93: Proceedings of the Winter 1993 USENIX Conference*, 1993.

[25] Patrick McDaniel, Shubho Sen, Oliver Spatscheck, Jacobus Van der Merwe, William Aiello, and Charles Kalmanek. Enterprise security: A community of interest based approach. In *NDSS'06: Proceedings of Network and Distributed Systems Security 2006*, 2006.

[26] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. In *Proceedings of 2003 IEEE Symposium on Security and Privacy*, 2003.

[27] David Moore, Colleen Shannon, and Jeffery Brown. Code-red: a case study on the spread and victims of an internet worm. In *USENIX'02: Proceedings of the 11th USENIX Security Symposium*, 2002.

[28] Robert Ricci, Jonathon Duerig, Pramod Sanaga, Daniel Gebhardt, Mike Hibler, Kevin Atkinson, Junxing Zhang, Sneha Kasera, and Jay Lepreau. The flexlab approach to realistic evaluation of networked systems. In *NSDI'07: Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, 2007.

[29] James E. Smith and Ravi Nair. *Virutal Machines: Versatile Platforms for Systems and Processes.* Elsevier Press, 2005.

[30] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the internet in your spare time. In *USENIX'02: Proceedings of the 11th USENIX Security Symposium*, 2002.

[31] Jamie Twycross and Matthew M. Williamson. Implementing and testing a virus throttle. In *USENIX'03: Proceedings of 12th USENIX Security Symposium*, 2003.

[32] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *OSDI'02: Proceedings of 5th Symposium on Operating Systems Designs and Implementation*, 2002.

[33] Nicholas Weaver, Vern Paxson, Stuart Staniford, and Robert Cunningham. An taxonomy of computer worms. In *WORM'03: Proceedings of the 1st ACM Workshop on Rapid Malcode*, 2003.

[34] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. In *OSDI'02: Proceedings of 5th Symposium on Operating Systems Designs and Implementation*, 2002.

[35] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI'02: Proceedings of 5th Symposium on Operating Systems Designs and Implementation*, 2002.

[36] Matthew M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In *ACSAC'02: Proceedings of the 18th Annual Computer Security Applications Conference*, 2002.

[37] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *SRDS'03: Proceedings of 22nd International Symposium on Reliable Distributed Systems*, 2003.