WormTerminator: An Effective Containment of Unknown and Polymorphic Fast Spreading Worms

Songqing Chen[1], Xinyuan Wang[2], Lei Liu[1], Xinwen Zhang[2], and Zhao Zhang[3]

[1]Department of Computer Science
George Mason University
Fairfax, VA 22030
{sqchen,lliu3}@cs.gmu.edu

[2]Department of Information and Software Engineering
George Mason University
Fairfax, VA 22030
{xwangc,xzhang6}@gmu.edu

[3]Department of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011
zzhang@iastate.edu

ABSTRACT

The fast spreading worm is becoming one of the most serious threats to today's networked information systems. A fast spreading worm could infect hundreds of thousands of hosts within a few minutes. In order to stop a fast spreading worm, we need the capability to detect and contain worms automatically in real-time. While signature based worm detection and containment are effective in detecting and containing known worms, they are inherently ineffective against previously unknown worms and polymorphic worms. Existing traffic anomaly pattern based approaches have the potential to detect and/or contain previously unknown and polymorphic worms, but they either impose too much constraint on normal traffic or allow too much infectious worm traffic to go out to the Internet before an unknown or polymorphic worm can be detected.

In this paper, we present WormTerminator, which can detect and completely contain, at least in theory, almost all fast spreading worms in real-time while blocking virtually no normal traffic. WormTerminator detects and contains the fast spreading worm based on its defining characteristic – a fast spreading worm will start to infect others as soon as it successfully infects one host. WormTerminator also exploits the observation that a fast spreading worm keeps exploiting the same set of vulnerabilities when infecting new machines. To prove the concept, we have implemented a prototype of WormTerminator and have examined its effectiveness against the real Internet worm Linux/Slapper.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive Software (e.g., viruses, worms, Trojan horses)*; C.2.5 [**Computer Communication Networks**]: Local and Wide Area Networks—*Internet*

ANCS'06, December 3–5, 2006, San Jose, California, USA.

General Terms

Security, Design, Experimentation

Keywords

WormTerminator, Zero-day Worms, Polymorphic Worms, Virtual Machine, Worm Containment

1. INTRODUCTION

The fast spreading worm (abbreviated as fast worm hereafter) is becoming one of the most serious threats to today's networked information systems that we are depending on daily. Unlike all other threats, such as virus, intrusions, and spyware, fast worms could automatically propagate themselves over the network to infect hundreds of thousands of hosts without user interactions and do great harm in a short time. For example, Slammer, whose size is only 376 bytes, has been observed to probe 4000 hosts per second on average and infected about 75,000 vulnerable hosts running Microsoft SQL in about 10 minutes [21]. Although Code Red I is slower, it doubled the infected population with 37 minutes or so and infected 360,000 Microsoft IIS servers.

What makes it really challenging to defend against a fast worm is its extremely fast propagation speed. In order to defend against a fast spreading worm, we need the capability to effectively detect and contain the worm automatically in real-time. To effectively contain a fast worm, we have to cut off its propagation link at the earliest possible time.

Existing worm containment strategies can be broadly classified into two categories: signature based and traffic pattern based. Signature based approaches [4, 14, 17, 18, 30, 31] are efficient and effective in detecting and containing known worms, but they are inherently ineffective against unknown worms and polymorphic worms [23]. Traffic pattern based approaches [25, 28, 36, 37] do not rely on the worm signature, but rather on the pattern of worm traffic. Since worm propagation does have very distinctive patterns, traffic pattern based approaches could potentially detect and contain previously unknown worms and polymorphic worms. However, traffic pattern based approaches can only detect and contain a worm after the worm has started its propagation. Existing traffic pattern based approaches (such as new connection limiting [37] or unique/failed connection number counting [25, 28]) either impose too much constraint on nor-

mal traffic or allow too much infectious worm traffic to go out. The former would greatly degrade the service quality provided by the protected machine, while the latter could lead to failure in containing fast worms, given the exponential nature of worm propagation [32].

Ideally, we want to be able to detect and contain all fast worms, whether or not they are previously unknown, whether or not they are polymorphic, and allow all the normal traffic at the same time. This requires the capability to accurately detect and contain any fast worm before it really propagates to other Internet hosts. In order to detect and contain previously unknown or polymorphic fast worms, we cannot rely on worm signatures. However, traffic pattern based approaches need to observe worm propagation traffic for some time before they can determine whether or not the outgoing traffic is worm propagation. In other words, to completely contain the propagation of any unknown worm, we need to detect its propagation. To detect the propagation of the unknown worm, we need to see the propagation of the worm. The key issue here is how to detect the propagation of any unknown worm before it propagates to and infects other Internet hosts.

In this paper, we present WormTerminator, which can detect and completely contain almost all fast spreading worms in real-time while blocking virtually no normal traffic. WormTerminator detects and contains fast worms based on their defining characteristic - a fast spreading worm will start to infect other hosts as soon as it successfully infects one host. Therefore WormTerminator could detect, at least in theory, all fast spreading worms. Unlike all previous worm detection and containment approaches, WormTerminator is able to detect the propagation of previously unknown or polymorphic fast worms before they can infect any other host. This is achieved by transparently diverting all outgoing traffic to a cloned virtual machine within the same host where WormTerminator resides. To the initiator of the traffic, the virtual machine appears to be the destination. WormTerminator exploits the observation that a worm keeps exploiting the same set of vulnerabilities as coded when infecting a new host. Therefore, if a worm has successfully infected the current host, it will successfully infect, after being diverted to, the virtual machine that has the exactly same vulnerabilities as the current host. Once the fast worm infects the virtual machine, the virtual machine will exhibit worm behaviors and start to infect other hosts. By monitoring the traffic pattern of the virtual machine for a specified period of time, WormTerminator is able to determine whether or not the diverted traffic is fast worm traffic without risking infecting other hosts. If the diverted traffic does not exhibit worm propagation behaviors, it will be forwarded to its real destination. In this case, the virtual machine acts as a transparent proxy between the traffic source and its original destination.

To prove the concept of WormTerminator, we have implemented a prototype in Linux and have examined its effectiveness against real Internet worm Linux/Slapper. Our empirical results confirm that WormTerminator is able to completely contain fast worm propagation while allowing virtually all normal traffic in real-time. The major performance cost of WormTerminator is a one-time delay to the start of each outgoing normal connection for worm detection. By utilizing cache techniques, on average WormTerminator will delay no more than 6% normal outgoing traffic.

The remainder of this paper is organized as follows. Section 2 briefly illustrates the need of effective worm containment schemes. Section 3 overviews the WormTerminator design. Section 4 discusses several design issues and our solutions. Section 5 describes our prototype implementation. Section 6 presents our experimental results on Linux/Slapper. Section 7 reviews related work. Finally, section 8 makes concluding remarks with future work.

2. EPIDEMIC MODEL OF FAST WORM PROPAGATION

Staniford *et al* proposed the random constant scan (RCS) worm propagation model. Given an initial compromise rate of K, along time t, the RCS model determines that the proportion of those vulnerable machines that have been compromised (denoted as α) is

$$\alpha = \frac{e^{K(t-T)}}{1 + e^{K(t-T)}}, \quad (1)$$

where T is a constant of integration that fixes the time position of the incident. The RCS model has been validated by the empirical propagation data of Slammer [20] with an initial compromise rate $K = 6.7$ per minute and $T = 1808.7$ second.

It is easy to see that even if the compromise rate K is reduced to 0.67 per minute, the time needed to compromise the vast majority of vulnerable hosts only increases about 15 minutes. This means that we have to keep the compromise rate K to a very low value in order to gain the time to react to the spreading of a fast worm. This is particularly challenging for those fast worms who scan with a hit-list (e.g., Warhol or flash worms [33]), which could make the compromise rate very close to the probe rate.

This indicates that simply throttling the worm probe traffic is neither effective in containing the worm nor acceptable to normal Internet applications. The dilemma here is how to block the worm traffic as much as possible while keeping the hosts open for normal network traffic. Ideally, we want to be able to contain all the fast worm traffic and allow all normal traffic at the same time. In the rest of paper, we show that it is possible to block all the probing traffic of previously unknown, polymorphic fast worms while allowing virtually all non-worm traffic at the same time.

3. OVERVIEW OF WORMTERMINATOR DESIGN

3.1 Design Goal and Principles

To completely contain fast worms, WormTerminator must examine and restrict outgoing traffic from the very beginning, i.e., the first exploit of a fast worm should be detected and stopped.

The main design goal of WormTerminator is to completely contain any known or unknown fast worm while allowing all non-worm traffic. In other words, we strive to detect and stop the first exploit from any fast spreading worm without blocking any non-worm traffic. To achieve such a design goal, we create a virtual machine that has cloned the operating system and server applications running on the host machine. This would allow us to detect the propagation of almost all fast worms before they can infect any other host

on the Internet. In addition, the virtual machine serves as a transparent proxy to all non-worm traffic.

The virtual machine clones the host operating system and server applications running on the host and is supposed to be started automatically by the host when it starts. The communication between the virtual machine and the host machine as well as other hosts on the Internet is controlled by the virtual machine monitor (VMM). In general, there are two types of VMM structures, depending on the relative positions of the VMM and the hardware [16]. Type I VMM always has the VMM running on the hardware, while Type II always has the VMM running on the host OS. WormTerminator can work with both types of VMM structures as long as the VMM is relatively well protected such that the infection of the host does not quickly compromise the VMM.

The principles underlying the WormTerminator design are as follows:

- **A worm always exploits the same set of vulnerabilities as coded.** Every worm is coded to exploit a certain set of vulnerabilities. Since the virtual machine is a clone of the host, it has the same vulnerabilities as the host. Therefore, if a worm has successfully exploited some vulnerabilities and has infected the current host, it is able to infect the virtual machine.
- **A fast worm always tries to propagate itself and infect others as soon as it has infected the current host.** This propagation behavior is the defining characteristics of fast worms, which makes the worm propagation traffic very distinct from other traffic. This unique traffic pattern is how we determine if any particular traffic is worm propagation traffic.

Based on these principles, WormTerminator does the following on any outgoing traffic from the host on which it resides:

- Transparently divert any outgoing traffic to the virtual machine for checking (worm detection);
- Monitor the traffic pattern of the virtual machine to determine if the diverted traffic is worm propagation;
- Forward the diverted traffic to its original destination once it is determined as non-worm traffic. The virtual machine starts to act as a transparent proxy for the original outgoing traffic;
- Drop any diverted traffic that has been determined to be worm propagation, take actions and report as appropriate.

By transparently diverting the outgoing traffic to the virtual machine, we are able to monitor any worm propagation behavior without risking infecting other hosts on the Internet. To the sender of the outgoing traffic, the virtual machine appears as the original destination. Therefore, if a fast worm is trying to propagate from the current host, its propagation traffic will reach the virtual machine no matter what destination it was trying to reach. Upon arriving at the virtual machine, the worm traffic will soon infect the virtual machine [1] and the virtual machine exhibits worm behaviors quickly. Therefore, the propagation of any fast worm will be detected and stopped at the virtual machine. On the other hand, normal traffic does not exhibit worm propagation behavior, thus it will be forwarded to the original destination eventually.

By examining the defining characteristics of worm propagation traffic in a carefully instrumented virtual machine, we are able to detect the propagation of fast worms at the very beginning and prevent the worm from infecting any other host on the Internet. At the same time, normal outgoing traffic is almost never blocked.

Compared with signature based worm detection and containment, WormTerminator is able to detect and completely contain previously unknown worms and polymorphic worms. Compared with existing traffic pattern based worm containment techniques, WormTerminator does not block any non-worm traffic, and completely blocks the infectious traffic from fast worms.

3.2 WormTerminator Architecture and Flow of Control

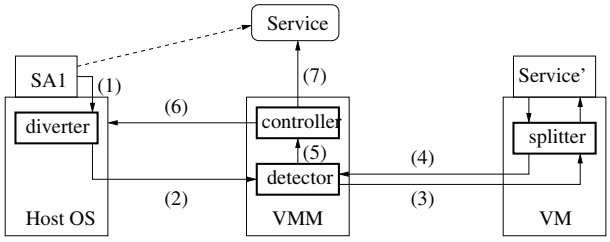

Figure 1: WormTerminator Architecture and Flow of Control

Figure 1 shows the architecture of WormTerminator and the typical flow of control for outgoing traffic. There are four major components in WormTerminator:

- *diverter*: it resides in the host OS, and is responsible for intercepting any application communication and sending it to the virtual machine through the VMM, pretending that the virtual machine is the destination.
- *detector*: it is located in the VMM. Once the VMM finds there is traffic to the virtual machine, it creates the environment, by setting up the IP of the virtual machine the same as the traffic destination, and opening corresponding ports if necessary. After preparation is done, the traffic is forwarded to the virtual machine, and the detector closely watches network behaviors of the virtual machine. If the forwarded traffic triggers any worm-like behavior, the detector will generate an alarm and report it to the controller. Otherwise, the detector will report the forwarded traffic as normal to the controller.
- *controller*: it logically resides in the VMM. Once it receives the report from the detector, the controller will either forward the normal traffic to its original destination or drop the worm traffic and raise an alarm to the user.
- *splitter*: it is running inside the virtual machine to duplicate the original request (packet). One request

[1] Unless the worm is able to determine if the host at which it arrives is a virtual machine and stops infection on a virtual machine. This case can be mitigated by instrumenting the physical host to appear as a virtual machine.

copy is sent to the local service for worm detection, and the other is kept in the local buffer in case it is normal traffic and should be sent to the real destination.

The four components collaborate with each other to achieve our design goal. As shown in figure 1, the server application SA1 needs to access an Internet service (indicated by the dashed line). However, the outgoing connection is not established directly, as would happen in a normal host. Instead, the diverter intercepts the outgoing packet and diverts it to the virtual machine through the VMM. Upon receiving the outgoing packet, the splitter at the virtual machine duplicates the request packet in its buffer before forwarding the request packet to the appropriate service running in the virtual machine. The detector monitors the network behavior of the virtual machine, determines whether the diverted request packet belongs to the worm propagation and reports the result to the controller in the VMM. The controller will forward any normal outgoing request packet to the original destination, and drop the worm propagation packet and report to the user.

4. DESIGN ISSUES AND SOLUTIONS

In this section, we discuss several important design issues and present our solutions.

4.1 How does WormTerminator detect the worm?

To stop the fast worm spreading, the worm must be detected at the earliest possible time. How to determine whether the traffic is worm propagation is one critical design issue. In principle, WormTerminator detects the worm by checking if the network traffic of the virtual machine has any worm propagation pattern. One simple criterion for detecting worm propagation pattern is timing correlation between incoming and outgoing traffic. The rationales behind using the timing correlation are the following: 1) fast worms strive to propagate to and infect as many other hosts as possible in the shortest possible time; 2) fast worms are usually small in size. Therefore, the volume of worm infecting traffic is small. After the fast worm traffic successfully infects a host, the infected host will start trying to infect other hosts in a short time. For example, we have observed that a Linux host will start sending out infectious traffic within 10 seconds after it is infected by Linux/Slapper worm.

Table 1: Trend of Worm Size

Name	Size	Year
Nimda	60 KB	2001
Code Red	4KB	2001
Slammer	376 bytes	2003

WormTerminator uses two time thresholds for detecting the propagation of fast worms. T_{time} is the maximum time interval between the time when the virtual machine receives the fast worm traffic and the time when the virtual machine starts to send out infectious traffic. T_{size} is the time needed to transfer the whole worm. As shown in Table 1, worms are getting smaller. Initially, we set T_{size} to be T_{100KB}, the time needed to transfer 100KB data since almost all fast worms are less than 100KB.

To detect if any traffic diverted to the virtual machine is worm traffic, the detector monitors network activities of the virtual machine. If the virtual machine receives some continuous traffic whose transmission time is less than T_{size}, and starts to send similar traffic to other hosts within time T_{time}, the diverted traffic is considered worm traffic. Here we do not count any traffic from the virtual machine to its host machine, and we only consider outgoing traffic from the virtual machine to other hosts on the Internet.

But how shall we determine T_{time}? This is critical for WormTerminator to *quickly* detect worms. It also affects how long an application needs to wait for worm detection. Ideally, T_{time} should be the time needed for a worm to complete its infection procedure. Clearly, different worms could take different time durations to complete such a procedure. Thus, there may not exist a fixed upper bound good for all. However, as Figure 2 shows, if both Host-A and Host-B have the same set of vulnerabilities that a worm exploits, the time interval I_1, for the worm to enter Host-A to the time Host-A becomes a source and starts to infect others, should be close to I_2, the time interval on Host-B for such a procedure, without considering the physical configuration differences between Host-A and Host-B. In the WormTerminator design, clearly, Host-A is the host, while Host-B is its virtual machine. Thus, if we can measure I_1, we can have a good estimate of I_2 and thus we can set up T_{time} accordingly.

Unfortunately, it is not easy to measure I_1. The difficulty lies in that on Host-A, there could be several multiple concurrent inbound network flows, although we are only interested in the one related to the flow to Host-B. Since normally worms exploit the vulnerability of a running process, from there a worm process is forked or the running process is hijacked, we thus can analyze the process information to determine which incoming flow is related to a particular outgoing flow. If the worm process is forked, through tracing its parent process we can get the information about when the parent starts the last communication. This information can be used to determine when the suspicious traffic enters Host-A, and thus I_1. If the process is hijacked, the related information can be directly extracted from the currently running process. However, applying process tracing to determine I_1 also needs to pay attention to the following exceptions. If the outgoing traffic to the virtual machine is not related to any incoming traffic to Host-A, e.g., it is caused by a user on Host-A, we assume that under this situation, the interval, I_1, is infinity. Considering that network level activities have timing constraints from the transport level, e.g., the network connection timeout, we also need to have a maximum threshold, `MAX_TIMEOUT`, for the waiting time. This `MAX_TIMEOUT` is OS dependent.

Consider the fact that the performance of a virtual machine is always slower than its original host. Denoting such slowness with a slowdown SD, we should turn $I_2 = SD \times I_1$. This leads to the final criteria, T_{time}, used in WormTerminator for worm detection if the transmission takes a time less than T_{size}:

$$I_2 = SD \times I_1,$$

$$T_{time} = min(I_2, \texttt{MAX\_TIMEOUT})$$

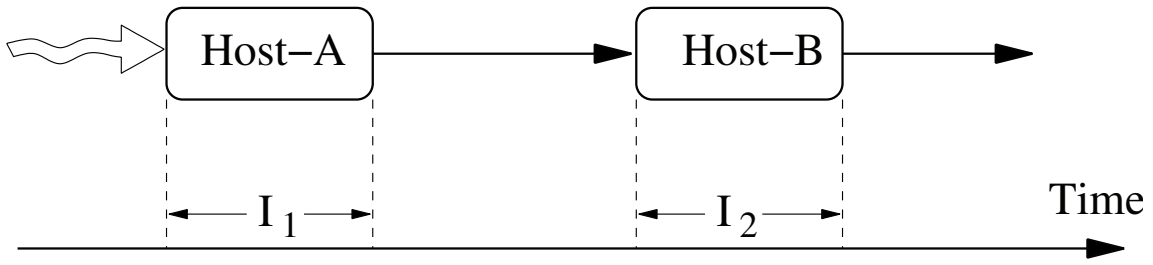

Figure 2: Detecting Worm Propagation Based On Timing Correlation. As shown in the figure, we denote as I_1 the interval between the time when the worm traffic gets into Host-A to the time when Host-A starts to send worm traffic to new hosts. If Host-B has the same set of vulnerabilities as A and is exploited by Host-A, without considering the physical speed and other configuration differences, it is expected that such an infection procedure should take a similar time duration, denoted as I_2 in this figure, on Host-B.

4.2 How does WormTerminator distinguish worm traffic from benign traffic with worm-like traffic pattern?

By definition, a fast spreading worm will start to infect others as soon as it successfully infects one host and thus will always be contained by WormTerminator. However, a few normal network applications may exhibit a similar traffic pattern as that of a fast worm, and special care is needed to differentiate such traffic from the worm traffic.

- **Email Relay**: To facilitate email transfer across the Internet, some SMTP servers function as relay in that they will forward the received email to the next SMTP server after adding some tracing information to the forwarded email. From outsider's point of view, this traffic pattern is similar to worm propagation. However, a normal email relay differs from worm propagation in three aspects. First, during the email relay, the SMTP server is not the final destination of the email. This is in contrast to the worm propagation where the infected host who is trying to infect others was indeed the destination of the infectious traffic that infected it. Second, normal email relay requires very little processing and it usually does not trigger noticeable system-wide actions. On the other hand, when a worm infects a host, it usually triggers noticeable system actions such creating a new process, reading or writing files, opening a new socket. Third, SMTP relay traffic always use port 25 while most fast spreading worms use other port numbers. Therefore, normal email relay traffic can be effectively differentiated from fast worm traffic. When a worm is propagated through email, it targets the email destination rather than the email relay hosts. In this case, WormTerminator could detect and contain the fast worm at the destination host of the malicious email.
- **P2P Search**: In some P2P applications like Gnutella, users frequently flood their queries. Normally a query receiver would pass the query to its neighbors if applicable (e.g., based on TTL). If the query receiver does not have the requested document, the traffic pattern of the receiver is similar to worm propagation. However, two features of P2P queries make them different from worm propagation. First, the size of P2P query is normally of tens of bytes while an unfragmented worm packet is unlikely to be less than 100 bytes. Second, a P2P query receiver only passes the query to its neighbors. In P2P networks, the neighbor information is always stored on the receiver when these neighbors joins the system, and such information is kept updated through some keepalive messages. Thus it is possible to distinguish P2P query flooding traffic by checking the packet size and keeping track of IP addresses of recent communications.
- **P2P Downloading**: Besides queries in P2P applications, some P2P downloading also exhibits a similar traffic pattern to that of worm propagation. For example, in BitTorrent-like systems, after a peer finishes downloading a file piece, it may simultaneously upload the file piece to several other peers. This traffic pattern is similar to that of worm propagation. The fundamental difference between the P2P downloading and worm traffic is that P2P downloading traffic normally follows a request-response model while worm traffic is almost always un-solicited. Therefore, we can differentiate the P2P downloading traffic from worm propagation traffic by checking if the current host is communicating with a host that has recently contacted the current host.

4.3 How does WormTerminator reduce the impact to normal applications?

In WormTerminator, in principle, all outgoing traffic is diverted to the virtual machine for checking (unless they are the applications mentioned above with a worm-like traffic pattern that are handled separately), which inevitably affects the original applications. Such impacts are in two folds. The first is transparency. That is, such traffic diversion should be made as transparent as possible to applications running on the host. The second is the performance. That is, the delay for worm detection to normal applications should be minimized. We discuss solutions to deal with them in detail as follows.

In terms of application transparency, while many applications (e.g., a browser) have built-in support for proxy, we cannot directly use it for diverting outgoing traffic. This is because the proxy is not the termination point, but a relay point. Since a worm is designed to infect the targeted host via an exploit on a particular application, it will not infect any proxy who merely relays the traffic to its ultimate destination. Therefore, we have to make sure the outgoing traffic

terminates at the virtual machine in order to let any worm traffic be able to infect the virtual machine. To achieve this, we can either change the destination IP address of the outgoing traffic to that of the virtual machine or dynamically set the IP address of the virtual machine to be the destination IP address of the outgoing traffic. Given that the outgoing traffic may have some built-in integrity check on the IP header (i.e. IPsec AH header), changing the destination IP address of outgoing traffic may not always feasible. Therefore, dynamically setting the IP address of the virtual machine is a better way to deceive worm traffic.

After setting the IP address of the virtual machine to be the destination IP address of the outgoing traffic, the virtual machine appears to be the destination of the outgoing traffic. After the diverted traffic terminates at the virtual machine, the detector decides whether the diverted traffic is worm traffic by monitoring the virtual machine's network activities for a specified period of time. If the diverted traffic is worm traffic, it will be blocked. Otherwise, it needs to be relayed to the real destination. For connectionless traffic such as UDP, we can simply forward the packets (saved by the splitter) from the virtual machine to its destination. For connection oriented traffic such as TCP, there is state information maintained at both sides of the communicating parties. To the sender on the host machine, the virtual machine is the destination. In this case, we can not simply forward the TCP packet to its destination. Instead, the virtual machine needs to reestablish a connection to the destination and starts to function as a relay or proxy between the sender in the host machine and the receiver on the real destination. The packets saved by the splitter are used for generating appropriate application level requests to be sent to the destination. In this sense, the virtual machine functions as an application aware proxy.

While we can make the operation of WormTerminator as transparent as possible to most applications on the host machine, there will be some extra overhead introduced by WormTerminator. To be specific, outgoing connections will be delayed when they are diverted to the virtual machine for checking. Several ways are possible to reduce the overall performance impact.

First, if some configurable number of UDP packets from some flow have passed checking, we can directly route the rest UDP packets of the same flow without diverting them to the virtual machine. This would decrease the average performance overhead of WormTerminator.

Another way to improve the performance of WormTerminator is to use a cache[2] to store such examined connections, and associate an expiration time with each cache entry. Before the expiration time, packets of recently examined connection will not be diverted to the virtual machine, but routed to its destination directly. For those connections that are not in the cache or have expired, the first configurable number of packets will be diverted to the virtual machine for checking. If they pass the checking, the connection will be put into the cache with an expiration time. Since normally client accesses show great temporal localities and spatial localities, this caching strategy can amortize the worm detection overhead over multiple repetitive connections.

These performance improvements represent some tradeoff between security and performance. Depending on the performance and security requirements, users of WormTerminator may choose to 1) divert all outgoing traffic to the virtual machine for checking; or 2) cache individual connection and only divert those packets that are not part of cached connections; or 3) cache all connections to a particular destination to which some connection has recently passed checking.

On the other hand, there is also a technology trend to put multiple, and possibly multithreaded, processor cores onto a single processor chip so as to fully utilize the available transistors and to tolerate very long memory latency. Most desktop/server processors today have more two processors cores; for example, Intel Pentium D and Core Duo 2, AMD Athlon Dual-core, IBM Power4 and Power5 [13], among many others. An extreme example is the Sun Niagara processor [24], which has eight 64-bit UltraSparc cores and each core can execute up to four threads, supporting 32 threads in total. Not all the time the applications may be able to fully utilize those cores and hardware thread contexts. On those processors, WormTerminator will be able to utilize idle cores or thread contexts, increasing the processor utilization and having less impact on the performance of the host system.

5. IMPLEMENTATION

To prove the concept of WormTerminator, we have implemented a prototype. To test with the Internet worm Linux/Slapper which attacks Apache servers, we have implemented the HTTP/HTTPS support in our prototype. In principle, WormTerminator could work with any application protocol if appropriate protocol support is added.

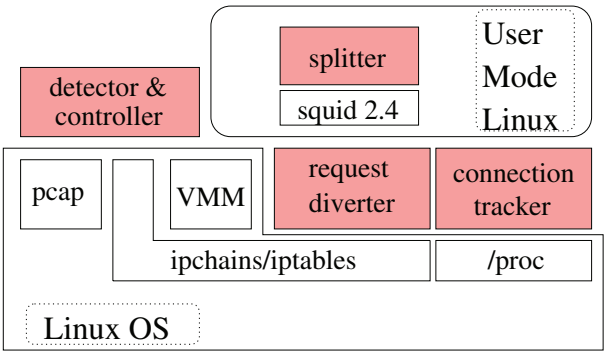

Figure 3: WormTerminator Prototype Implementation

Figure 3 shows the modularized implementation. Our implemented components are shown in shadow. The host OS is RedHat 7.3, running Linux kernel 2.4.18. The virtual machine we use is User-Mode-Linux [9]. As implied by the name, User-Mode-Linux itself runs as an application process in the host OS. The disk storage for User-Mode Linux is contained entirely inside a single file on the host machine, called the root filesystem for User-Mode-Linux. It provides several approaches to supporting virtual machine communications with the host physical machine and the world outside.

As shown in this figure, there are four major parts in our prototype implementation.

- connection tracker: who traces the incoming and out-

[2]This cache can be combined with the cache used to address the worm-like benign traffic.

going connection flows to and from the host. The purpose of such a component is to determine I_1 and thus set up I_2. It is implemented as a kernel module on the `/proc` filesystem of the host machine.

- request diverter: who captures and diverts all client requests to User-Mode-Linux. It is implemented as a kernel module hooked to `ipchains/iptables` on the host machine.

- splitter: who duplicates and stores application level requests from the traffic diverted to the virtual machine. It is implemented based on Squid 2.4STABLE1 (with cache function disabled) and runs inside of User-Mode-Linux.

- detector and controller: they are implemented in one daemon to monitor the traffic and make the examination decision with the help of the `pcap` library, `ipchains/iptables`, and the `VMM`. A host TUN/TAP device is used for User-Mode-Linux communications [9].

We have also ported our prototype implementation to Fedoral Core 2 with kernel 2.6.5. We are in the process of adding more protocol support to our Linux prototype, and we plan to implement WormTerminator on Windows platforms.

6. EVALUATIONS

In this section, we empirically evaluate WormTerminator and seek to answer the following questions: 1) how effective the is WormTerminator in containing real worm propagation traffic mingled with normal traffic? and 2) what is the impact to normal applications? Due to page limit, we omit the overhead measurement of WormTerminator.

6.1 Linux/Slapper Test

Linux/Slapper [1] is a family of worms exploiting the vulnerability of an `OpenSSL` buffer overflow in the `libssl` library, which further enables Distributed Denial of Service (DDoS) attacks [3]. It is different from many existing worms since it targets the buffer overflow in the heap.

Slapper targets vulnerable Apache Web server 1.3 on Linux operating systems, including `RedHat`, `SuSe`, `Mandrake`, `Slackware`, and `Debian`. According to Symantec DeepSight Threat Management System, more than 3500 computers were infected [2].

The basic procedure that Slapper uses is as follows. When a worm instance is active, it scans class-B networks, looking for Apache servers by attempting to connect to port 80. After determining the server is vulnerable, it tries to send the exploit code to the SSL service via port 443. Upon an successful exploit, Slapper encodes its source code (`.bugtraq.c`) and sends to the victim and stores as a hidden file (`.uubugtraq`) under `/tmp`. There, it uu-decodes the file, compiles, and executes the binary, with the sender's address as an input parameter.

The exploit procedure of Slapper is more complicated than many existing fast worms. A successful exploit uses buffer overflow twice, and takes 1+20+2 requests. The first one is used to get the Apache server version information. The next 20 are used to force Apache to use up possible existing processes. Then two HTTPS requests are launched to exploit the vulnerability and inject the shell code, upload itself, compile and execute the binary.

Compared to worms that we have listed in Table 1, the size of Slapper is also large[3]. The original source code is 67655 bytes, and the uu-encoded source code is propagated between vulnerable hosts, which is 93461 bytes.

To test whether this worm can be successfully contained by WormTerminator, we set up our environment as follows. The host runs `RedHat 7.3`, with `Apache 1.3.23`, `mod_ssl 2.8.6`, and `OpenSSL 0.9.6`. The kernel is 2.4.18. User-Mode-Linux has the same configurations. The machine is running with a 2.4 GHz CPU and 1 GB physical memory.

Two other machines are set up in the same local network with the same configurations, connected through a 10/100 M hub. One is acting as the Slapper original source with 127.0.0.1 as the input parameter, and the other is the trigger with the IP address of the first as the input parameter. We slightly change the source code so that the worm starts to exploit the network segment where the host resides without waiting to exploit other non-related network addresses first as originally coded.

For the effectiveness experiments, the MAX_TIMEOUT is set as 2 minutes, default by TCP. The other important parameter is SD, which is critical depending on the performance slowdown of the virtual machine. Thoroughly studying the performance slowdown of any virtual machine is not the focus of this study. However, a previous study [16] has reported that compiling Linux 2.4.18 kernel inside UML-Linux [5] takes 18 times as long as compiling it on a Linux host operating system. Considering that there is few network activities involved in kernel compiling and User-Mode-Linux is faster than UMLinux, we setup SD for our User-Mode-Linux with 18 too. In our experiments, currently T_{size} is T_{100KB}.

We run the experiments 10 times, and each time WormTerminator successfully captured Slapper at the worm's first exploit. Table 2 shows our measurement results with the average and the standard deviation. The small standard deviation indicates the consistency of measurement results. A successful infection only takes about 10 seconds between physical machines. To verify this, we also instruct the worm source code directly and get very close results. It takes about 1.5 minutes to make the detection decision, which implies a slowdown of User-Mode-Linux around 10. The code transmission time differences indicate that the network transmission speed is only roughly half of the physical link. We will further evaluate this overhead in the next subsection.

Table 2: Slapper: infection and code transmission time (second)

	I_1		I_2	
	infection	code Xfer	infection	code Xfer
average	9.3456	3.0654	91.8893	6.9773
std_dev	0.4666	0.0120	1.2896	0.1103

From the experimental results, we can see if the performance of User-Mode-Linux is better with a smaller slowdown, the detection time could be further reduced.

[3]Slapper first appeared in 2002.

Table 3: Web sites used to test false positive and false negative

Protocol	Web Site	Activities
HTTP	www.cnn.com	Browsing
HTTP	www.usatoday.com	Browsing
HTTP	www.acm.org	Browsing
HTTPS	gmail.com	Email access
HTTPS	www.discovercard.com	E-transactions

To study whether we can detect worms in a mix of traffic, i.e., false positives and false negatives, we perform the following two sets of experiments. In the first set, with a normal Mozilla browser (version 0.9.9), a few Web sites as listed in Table 3 are accessed repetitively from the host machine to test if WormTerminator would falsely take any traffic as worm traffic. In all the experiments, the Squid cache function is disabled. In the period of our experiments of 1 hour, no false positive is found. In the second set of experiments, while these Web sites are accessed, Slapper is activated. In all cases, WormTerminator successfully detects the worm traffic at its first exploit.

The inadequacy of these tests is that we only have one real Linux worm. We are working to get more Linux worm source codes and plan to test when more worms are available and active together.

6.2 Impact on Normal Applications

With a cache in WormTerminator, some client traffic could avoid being examined and thus do not suffer the long delay. To enable this function, the examined connections must be saved in the cache.

As mentioned above, there could be different levels of cache. The object for caching could be the connection (destination host and port), or could be the host alone. For HTTP/HTTPS requests, we can even cache the client request.

For different levels of caches, different sizes of cache space are required. To study how many client requests would be affected with a what size of the cache, we run a simple simulator to analyze six client Web browser logs collected in a lab environment for about 4 months. Table 4 briefly summarizes some statistics of client access logs.

Table 4: Client Log Statistics

	#requests	#requests (unique)	#connections (unique)
client1	8318	2130	362
client2	12852	2724	455
client3	8921	1843	289
client4	7809	2074	337
client5	24793	5789	1119
client6	8457	2179	381

First, we consider to use cache to cache client requests. Following the idea of Squid, caching of one request demands a memory size of 128 bits after applying MD5 to the URL. With the field of expiration time, each request cache entry is 20 bytes. Note that in our simulations, the expiration time is not used and the replacement is purely based on LRU.

Figure 4 shows the performance of the request cache when the cache size increases. The figure shows that when the cached objects are requests, a size of 64 cache entries (equivalent to 1.25-KB memory size) is good enough to achieve near optimal performance. A 1.25-KB memory is a trivial cost for modern computers. However, with a request cache, roughly 28% of requests have to be examined, and thus suffer a long delay due to worm detection in WormTerminator.

To further decrease this ratio and improve the client performance, we also consider to cache the connection. One connection cache entry includes destination IP, port, and expiration time, which requires 10 bytes.

Figure 5 shows the connection cache performance with a LRU cache replacement policy. As the figure indicates, a cache with 8 units (equivalent to 80 bytes) is good enough to approximately achieve optimal performance. Thus, if a connection cache is used, the cost is very trivial, and less than 6% of client requests suffer the long delay caused by the worm detection processing in WormTerminator. Our examination of a host cache gives similar results as the connection cache, because Web servers normally use fixed ports.

The cache performance is largely determined by client access locality. The above experiments are just case studies to demonstrate that different levels of caches can mitigate the impact of WormTerminator on normal applications. A more sophisticated cache can apply some advanced replacement policy and consider expiration time. We leave that for our future work.

7. RELATED WORK

Internet worm defense has been a long term problem. Both passive defending approaches and active defending approaches have been extensively studied. Passive approaches basically restrict incoming traffic, e.g., through firewalls, while active approaches restrict outgoing traffic. Compared with passive approaches, with which worm traffic still flows on the Internet, active approaches can limit worm traffic to the Internet and thus mitigate the worm traffic disturbance to the Internet. In addition, passive approaches, such as firewalls, are always vulnerable to evasion opportunities [34]. Whether an active or a passive approach is taken, the worm must be detected in the first place. The worm detection strategies currently used basically fall into two categories.

The first is signature based. Generating a content-based signature is a traditional approach. As the worm spreads very fast today, automatic systems have been proposed to generate worm signatures [14, 17, 31]. Since application messages may be scattered over multiple packets, fast signature extraction algorithms have been proposed in EarlyBird [30] and Autograph [14]. However, it is difficult for such an approach to detect unknown worms or fast worms that spread extremely fast and leave no time for human-mediated response. The polymorphic worms or encrypted worms further challenge its capability. Compared with Polygraph [23], Hamsa [18] is shown to be able to improve the speed, accuracy, and attack resilience of fast signature generation for zero-day polymorphic worms. It has been shown that Polygraph is vulnerable to deliberate noise injection [26]. Shield [35], instead of directly dealing with worms, generates vulnerability based filters to prevent possible vulnerability exploits. Similar to the fact that not all users are willing to patch their systems in time due to various reasons, users may not get these filters on in time. In addition, if the

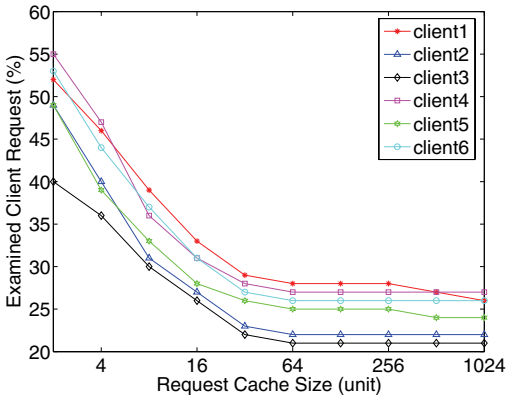

Figure 4: Request Cache Effect: the portion of client requests that is affected. Note the x-axis is in log scale.

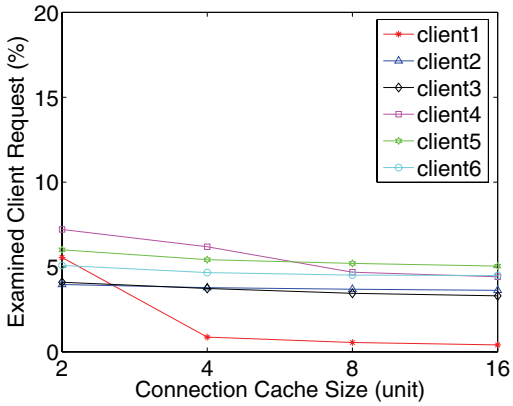

Figure 5: Connection Cache Effect: the portion of client requests that is affected. Note the x-axis is in log scale.

attack targets some vulnerability that has not been discovered before, Shield is not capable of generating such filters. A recent work [4] has focused on the automatic vulnerability signature generation with a single sample exploit, which is of much higher quality than exploit-based signatures.

Without relying on worm content, the second approach is based on the observation or analysis of network traffic. If some abnormal traffic pattern is found, the reaction system is triggered to take actions, such as blocking connections to some ports or limiting the rate of outgoing connections. Since worms scan as many vulnerable hosts as possible, Snort [28] monitors the connection rate to unique IP addresses. Because random scanning is likely to be rejected with a high probability, Bro [25] monitors the failed connection numbers while the failed connection rate is collected in work [36]. For reliable detection, traffic normalizers [11, 29] or protocol scrubbers [19] have been proposed to protect the forwarding path by eliminating potential ambiguities before the traffic is seen by the monitor. Work [37] proposes a heuristic strategy that limits the rate of connections to new hosts, e.g., to allow one new connection in a second. The system proposed in [36] targets one scan per minute of compromised hosts. More broadly, some other attack detection and signature extraction rely on the honeypots that cover dark or unused IP addresses, such as Backscatter [22], honeyd [27], honeyComb [17], and HoneyStat [8]. Any unsolicited outgoing traffic from the honeypots reveals the occurrence of attacks.

Recently, a number of works have been based on virtual machine technology to deal with various security problems, including intrusion detection [6, 10], vulnerability validation [7, 12]. Notably work [15] has examined security issue of the virtual machine itself.

While there are a number works utilizing the virtual machine technology to catch worms and study worm behavior, our work is the first, to the best of our knowledge, to leverage a virtual machine to contain the propagation of fast worms.

8. CONCLUSION

Detecting and containing fast spreading worms in real-time are very challenging, especially for those previously unknown or polymorphic worms. The key contribution of this paper is that we have demonstrated that it is indeed possible to detect and contain almost all unknown, polymorphic worms in real-time while allowing virtually all normal traffic to go out.

Our worm detection and containment are based on the defining characteristic of fast worms. By leveraging the virtual machine technology, we are able to detect the propagation of any fast worm before it can infect any other host on the Internet. This would allow us to almost completely contain almost all fast worms no matter whether they are unknown, polymorphic or not. We have validated our WormTerminator concept by implementing a prototype in Linux, and have examined its effectiveness against real Internet worm Linux/Slapper. Our real-time experiments confirm that our WormTerminator is able to contain fast worms without blocking normal traffic. We are currently optimizing the performance of WormTerminator.

Acknowledgment

We would like to thank Bill Bynum and the anonymous reviewers for their helpful comments. The work is partially supported by NSF grants CNS-0509061, CNS-0524286, CCF-0541366, and CNS-0621631.

9. REFERENCES

[1] `http://www.symantec.com/avcenter/venc/data/linux.slapper.worm.html`.

[2] `http://www.symantec.com/index.htm`.

[3] An analysis of the slapper worm exploit. `http://www.symantec.com/avcenter/reference/analysis.slapper.worm.pdf`.

[4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of IEEE Symposium on Security and Privacy*, Berkeley/Oakland, CA, May 2006.

[5] K. Buchacker and V. Sieh. Framework for testing the fault-tolerance of systems including os and network aspects. In *Proceeding s of the IEEE Symposium on*

High Assurance System Engineering (HASE), pages 95–105, October 2001.

[6] P. Chen and B. Boble. When virtual is better than real. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, pages 133–138, May 2001.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proceedings of SOSP*, Brighton, United Kingdom, October 2005.

[8] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. Honeystat: Local worm detection using honeypots. In *Proceedings of RAID*, 2004.

[9] J. Dike. A user-mode port of the linux kernel. In *Proceedings of the Linux Showcase and Conference*, October 2000.

[10] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, pages 211–224, December 2002.

[11] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of USENIX security Symposium*, August 2001.

[12] A. Joshi, S. King, G. Dunlap, and P. Chen. Detecting past and present intrusion through vulnerability-specific predicates. In *Proceedings of SOSP*, Brighton, United Kingdom, October 2005.

[13] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, March/April 2004.

[14] H. Kim and B. Karp. Autograph: Toward automated distributed worm signature detection. In *Proceedings of USENIX Security*, San Diego, CA, August 2004.

[15] S. King, P. Chen, Y. Wang, C. Verbowski, H. Wang, and J. Lorch. Subvirt: Implementing malware with virtual machines. In *Proceedings of IEEE symposium on security and privacy*, Berkeley/Oakland, CA, May 2006.

[16] S. King, G. Dunlap, and P. Chen. Operating system support for virtual machines. In *Proceedings of the Annual USENIX Technical Conference*, June 2003.

[17] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *Proceedings of HotNets*, Boston, MA, November 2003.

[18] Z. Li, M. Sanghi, Y. Chen, M. Kao, and B. Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of IEEE Symposium on Security and Privacy*, Berkeley/Oakland, CA, May 2006.

[19] G. Malan, D. Watson, and F. Jahanian. Transport and application protocol scrubbing. In *Proceedings of IEEE INFOCOM*, 2001.

[20] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the sapphire/slammer worm. `http://www.caida.org/publications/papers/2003/sapphire/sapphire.html`.

[21] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. In *Proceedings of IEEE Security and Privacy*, volume 1, July 2003.

[22] D. Moore, C. Shannon, and Jeffery Brown. Code-red: a case study on the spread and victims of an internet worm. In *Proceedings of the second Internet Measurement Workshop*, November 2002.

[23] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, CA, May 2005.

[24] K. Aingaran P. Kongetira and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2), 2005.

[25] V. Paxson. Bro: a system for detecting network intruders in real time. In *Computer Networks*, volume 31, December 1999.

[26] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. Sharif. Misleading worm signature generators using deliberate noise injection. In *Proceedings of IEEE symposium on security and privacy*, Berkeley/Oakland, CA, May 2006.

[27] N. Provos. A virtual honeypot framework. Technical report, University of Michigan, October 2003.

[28] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of Conference on System Administration*, November 1999.

[29] U. Shenkar and V. Paxson. Active mapping: Resisting nids evasion without altering traffic. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2003.

[30] S. Singh, C. Estan, G. Varghese, and S. Savage. The earlybird system for real-time detection of unknown worms. Technical report, University of California, San Diego, August 2003.

[31] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of OSDI*, San Francisco, CA, December 2004.

[32] S. Staniford. Containment of scanning worms in enterprise networks. In *Journal of Computer Security*, 2004.

[33] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time. In *Proceedings of USENIX Security*, San Francisco, CA, August 2002.

[34] t. Ptacek and T. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. `http://www.insecure.org/stf/secnet-ids/secnet-ids.html`, January 1998.

[35] H. Wang, C. Guo, D. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *Proceedings of ACM SIGCOMM*, Portland, OR, August 2004.

[36] N. Weaver, B. Staniford, and V. Paxson. Very fast containment of scanning worms. In *Proceedings of USENIX Security*, San Diego, CA, August 2004.

[37] M. Williamnson. Throttling viruses: Restricting propagation to defeat mobile malicious code. In *Proceedings of Annual Computer Security Applications Conference*, Las Vegas, NV, December 2002.