A Host-Based Approach for Unknown Fast-Spreading Worm Detection and Containment

SONGQING CHEN, LEI LIU, and XINYUAN WANG, George Mason University
XINWEN ZHANG, Samsung Information Systems America
ZHAO ZHANG, Iowa State University

The fast-spreading worm, which immediately propagates itself after a successful infection, is becoming one of the most serious threats to today's networked information systems. In this article, we present WormTerminator, a host-based solution for fast Internet worm detection and containment with the assistance of virtual machine techniques based on the fast-worm defining characteristic. In WormTerminator, a virtual machine cloning the host OS runs in parallel to the host OS. Thus, the virtual machine has the same set of vulnerabilities as the host. Any outgoing traffic from the host is diverted through the virtual machine. If the outgoing traffic from the host is for fast worm propagation, the virtual machine should be infected and will exhibit worm propagation pattern very quickly because a fast-spreading worm will start to propagate as soon as it successfully infects a host. To prove the concept, we have implemented a prototype of WormTerminator and have examined its effectiveness against the real Internet worm Linux/Slapper. Our empirical results confirm that WormTerminator is able to completely contain worm propagation in real-time without blocking any non-worm traffic. The major performance cost of WormTerminator is a one-time delay to the start of each outgoing normal connection for worm detection. To reduce the performance overhead, caching is utilized, through which WormTerminator will delay no more than 6% normal outgoing traffic for such detection on average.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software (e.g., viruses, worms, Trojan horses)*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

General Terms: Design, Security

Additional Key Words and Phrases: WormTerminator, zero-day worms, polymorphic worms, virtual machine, worm containment

ACM Reference Format:
Chen, S., Liu, L., Wang, X., Zhang, X., and Zhang, Z. 2014. A host-based approach for unknown fast-spreading worm detection and containment. *ACM Trans. Auton. Adapt. Syst.* 8, 4, Article 21 (January 2014), 18 pages.
DOI:http://dx.doi.org/10.1145/2555615

L. Liu is currently affiliated with Vuclip.
X. Zhang is currently affiliated with Samsung Telecommunications America.
This work was partially supported by the US NSF grants CNS-0509061, CNS-0746649, and CNS-1117300, and the US AFOSR grant FA9550-09-1-0071.
An earlier version of this work was published as "Wormterminator: An effective containment of unknown and polymorphic fast spreading worms" in *Proceedings of the 2nd ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'06)*. 173–182.
Authors' addresses: S. Chen and X. Wang, Department of Computer Science, George Mason University, Fairfax, VA 22030; email: {sqchen, xwangc}@cs.gmu.edu; L. Liu, Vuclip, 1551 Maccarthy Blvd. #213, Milpitas, CA 95035; email: lliu@vuclip.com; X. Zhang, Mobile Communication Lab, Samsung Telecommunications America, Santa Clara, CA 95054; email: xinwen1.z@samsung.com; Z. Zhang, Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011; email: zzhang@iastate.edu.

1. INTRODUCTION

The fast-spreading worm (abbreviated as fast worm hereafter), which immediately propagates itself after a successful infection, 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 [Moore et al. 2003]. Although Code Red I is slower, it doubled the infected population with 37 minutes or so and infected 360,000 Microsoft IIS servers.

Existing worm containment strategies can be broadly classified into two categories: signature based and traffic pattern based. Signature based approaches [Brumley et al. 2006; Kim and Karp 2004; Kreibich and Crowcroft 2003; Li et al. 2006; Singh et al. 2003, 2004] are efficient and effective in detecting and containing known worms, but they are inherently ineffective against unknown worms and polymorphic worms [Perdisci et al. 2006]. Traffic pattern based approaches [Paxson 1999; Roesch 1999; Weaver et al. 2004; Williamson 2002] 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 [Williamson 2002] or unique/failed connection number counting [Paxson 1999; Roesch 1999]) either impose too many constraints on normal traffic or still allow 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 the fast worms that are not scanning worms, given the exponential nature of worm propagation [Staniford 2004].

Ideally, we want to be able to detect and contain all unknown and polymorphic fast worms without affecting normal traffic. For this objective, in this article, we present WormTerminator, which can detect and completely contain almost all fast spreading worms in real-time without blocking any non-worm traffic. WormTerminator provides a host-based solution for fast Internet worm detection and containment with the assistance of virtual machine techniques based on the fast worm defining characteristic. In WormTerminator, a virtual machine cloning the host OS runs in parallel to the host OS and WormTerminator exploits the observation that a worm keeps exploiting the same set of vulnerabilities as coded when infecting a new host. With WormTerminator, any outgoing traffic from the host is diverted through the virtual machine. To the initiator of the traffic, the virtual machine appears to be the destination. Should the traffic be for fast worm propagation, the virtual machine will exhibit worm propagation pattern very quickly because a fast spreading worm will start to propagate to and infect others as soon as it successfully infects a machine. Therefore, if a worm has successfully infected the current host, it will successfully infect, after being diverted to, the virtual machine, which 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 incoming and outgoing traffic patterns 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. Therefore, 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.

To prove the concept of WormTerminator, we have implemented a prototype and have examined its effectiveness against real Internet worm Slapper. Our empirical results confirm that WormTerminator is able to completely contain worm propagation in real-time without blocking any normal traffic. The major performance cost of WormTerminator is a one-time delay to the start of each outgoing normal connection for worm detection, which mainly originates from the performance slowdown of the current virtual machine. In order to reduce such overhead, WormTerminator can leverage caching techniques, through which, WormTerminator will delay no more than 6% normal outgoing traffic for such detection on average.

WormTerminator is a host-based fast worm containment system. It automatically diverts outgoing traffic to the virtual machine for worm detection. Since different fast worms may take different durations for propagation, it is difficult to set an appropriate time duration for examining the outgoing traffic. Instead of setting up a fixed examination period – commonly used in the industry, WormTerminator adaptively sets up the detection period based on the observation on the host. Such adaptiveness would greatly reduce the false negatives generated otherwise. WormTerminator independently differentiates the traffic and makes the decision on whether to block or allow the outgoing traffic after the detection. The detection and the following operations are autonomous. Furthermore, with the assistance of cache, it automatically filters out benign traffic for improved user performance.

The remainder of this article is organized as follows. Section 2 overviews the WormTerminator design. Section 3 discusses several design issues and our solutions. Section 4 describes our prototype implementation. Section 5 presents our experimental results on Linux/Slapper and overhead measurements of WormTerminator. Section 6 further discusses some limitations and possible optimizations. Finally, Section 7 makes concluding remarks with future work.

2. OVERVIEW OF WORMTERMINATOR DESIGN

2.1. Design Goal and Principles

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 service 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 for all non-worm traffic. The virtual machine 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).

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

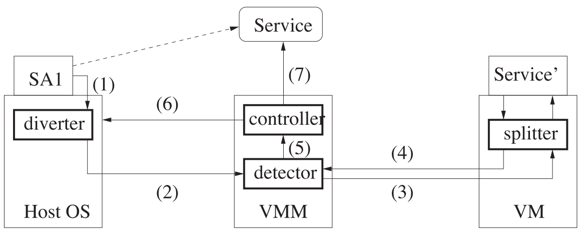

Fig. 1. WormTerminator architecture and flow of control.

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 for worm propagation.

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

— It transparently diverts any outgoing traffic to the virtual machine for checking (worm detection).
— It monitors the traffic pattern of the virtual machine to determine if the diverted traffic is worm propagation.
— It forwards 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.
— It drops any diverted traffic that has been determined for worm propagation, take actions and report as appropriate.

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 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.

2.2. 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 necessary environment and 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. One request 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 service 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 packets and diverts them to the virtual machine through the VMM. Upon receiving the outgoing packets, the splitter at the virtual machine duplicates the request packets in its buffer before forwarding the request to the appropriate service running in the virtual machine. The detector monitors the network behavior of the virtual machine, determines whether the diverted request 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.

3. DESIGN ISSUES AND SOLUTIONS

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

3.1. How Does WormTerminator Detect the Worm?

To stop worm spreading, the worm must be detected at the first opportunity. 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 criteria 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. For example, VBS/Loveletter (2000) is about 10 KB and Nimda (2001) is about 60 KB. While Ramen (2001) and Code Red (2001) are 7 KB and 4 KB, respectively, Slammer (2003) and Witty (2004) are only 376 bytes and 1184 bytes, respectively. 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.

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 worms are getting smaller in size, 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.

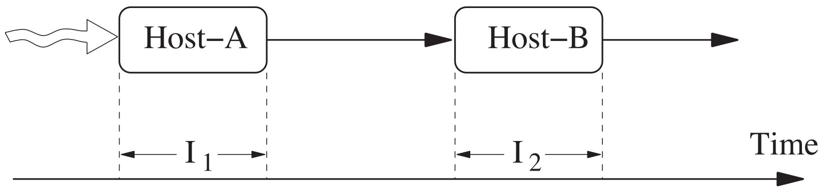

Fig. 2. Detecting worm propagation based on timing correlation. As shown in the figure, we denote as I_1 the interval between the time when the worm traffic gets into Host-A to the time when Host-A starts to send worm traffic to new hosts. If Host-B has the same set of vulnerabilities as A and is exploited 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.

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. In the WormTerminator design, clearly, Host-A is the host, and 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. We thus need to keep track of connections from the beginning. Since normally worms exploit the vulnerability of a running process, from which a worm process is forked or the running process is hijacked, we thus can 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 started the last communication.[1] 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. If the outgoing traffic to the virtual machine is not related to any incoming traffic to Host-A, for example, 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 at the transport level, for example, the network connection timeout, we also need to have a maximum threshold, `MAX_TIMEOUT`, for the waiting time. This `MAX_TIMEOUT` is OS dependent.

In practice, 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}:

$$I_2 = SD \times I_1,$$

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

[1]In practice, we can trace the time when Host-A and Host-B receive their last packets. In addition, we need to track each flow from the beginning since when we want to find the related flow information in the parent process in Host-A, the parent process may have exited.

3.2. How Does WormTerminator Distinguish Worm Traffic from Benign Traffic with Worm-like Traffic Pattern?

According to the criteria described here, if there is fast worm traffic, it will spread to other machines as soon as possible after it is diverted to the virtual machine. Although worm propagation does have this traffic pattern (and thus will always be contained by WormTerminator), in practice, some benign traffic may have this worm-like traffic pattern, for example, the e-mail relay service, some peer-to-peer (P2P) applications. Directly applying the basic criteria may cause high false positives. Thus, in the current design of WormTerminator, we integrate the following solutions to deal with three typical benign traffic that may expose similar traffic patterns as fast worm propagation.

3.2.1. E-mail Relay. SMTP servers are common on the Internet. One of the functions that most SMTP servers support is e-mail relay for various reasons. Such e-mail relay pattern is quite similar to worm propagation. That is, the SMTP server receives an incoming e-mail, without changing any content but only adding some tracing information, and forwards to the next SMTP server or servers.

To deal correctly with such e-mail relay, our solution is designed based on the fundamental difference between e-mail relay and worm propagation. For e-mail relay, the relay server is not the final destination of the e-mail. If the traffic is for fast worm propagation, the server is the destination. Such information is not directly available at the network level, but could be extracted from the application header. As SMTP always uses port 25, it is thus easy to distinguish them through header interpretation. This approach, however, sometimes may suffer from end-to-end encryption and other specific application-level processing, such as spam assassination. Under such situations, we need to rely on more heuristics. For example, when the e-mail is encrypted, according to the SMTP protocol, e-mail relay involves almost no processing at the relay server, while worm traffic, will execute on the machine before it propagates to other hosts. Thus, it is possible to monitor the behaviors (such as whether a new process is forked, where there is nontrivial changes of memory and CPU usage) of the host after receiving the traffic.

Similar to e-mail relay, various Proxy servers, such as for http and socks, exhibit similar patterns as E-mail Relay servers, and we can use similar heuristics as mentioned here to deal with them.

3.2.2. P2P Search. In some P2P applications, such as BearShare, LimeWire, Morpheus of Gnutella and KaZZa, it is frequent that a user floods his query. Normally a receiver would pass this query to its neighbors if applicable (such as based on TTL), whether this peer has the requested document or not. If this receiver does not have the document for the query, the traffic pattern of the receive is similar to worm propagation too. However, there are two distinct features that make such P2P queries different from worm propagation. The first is that a receiver only passes to its neighbors. In P2P networks, the neighbor information is always stored on the receiver when these neighbors joined the system, and such information is kept updated by communicating through some keepalive messages. Thus, in WormTerminator, if the IP addresses of these recent communication parties are recorded (and updated accordingly), WormTerminator can easily distinguish it from worm traffic propagation. In case that the worm traffic also leverages some address information on the host to propagate, the second characteristic we utilize is the size of the traffic. For fast worm, it is less likely the traffic is smaller than 100 bytes. While for common P2P queries, the size of the query is normally in tens of bytes.

3.2.3. P2P Downloading. Besides queries in P2P applications, P2P downloading also exhibits a similar traffic pattern to that of worm propagation. For example, in

Bit-Torrent like systems, after a peer finishes downloading a file piece, it may simultaneously upload to several other peers, a pattern that worm propagation also exhibits. However, a fundamental difference that enables us to differentiate them is that: worm propagation traffic is always unsolicited, while P2P downloading traffic always follows a request-response model. That is, in P2P downloading, unless a peer contacts another for a file piece, the peer will not actively upload this file piece to any other. Thus, an uploading must have been resulted from a previous request. If a temporal cache space could be used to store the IP addresses that have communicated with the machine recently, the P2P downloading traffic can be easily filtered out.

3.3. How Can WormTerminator Reduce the Impact on Normal Applications?

In WormTerminator, in principle, all outgoing traffic is diverted to the virtual machine for checking, which inevitably affects the original applications. Such impacts are twofolds. 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.

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. In addition, a worm can easily detect whether it is trapped by comparing its destination address and the exploited host's IP address. 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 or not 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, the virtual machine can simply forward the traffic (saved by the splitter). For connection-oriented traffic such as TCP, 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 traffic saved by the splitter is 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.

To minimize the delay impact, first, in order to reduce overheads in naive UDP streaming protocols such as multimedia broadcasting, 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 scheme to improve the performance of WormTerminator is to use a cache to store such examined connections, and associate an expiration time with each cache

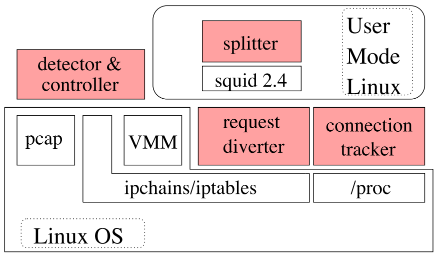

Fig. 3. WormTerminator prototype implementation.

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.

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 [Kalla et al. 2004], among many others. An extreme example is the Sun Niagara processor [Kongetira et al. 2005], 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.

4. 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.

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. We have also ported our prototype implementation to Fedoral Core 2 with kernel 2.6.5. The virtual machine we use is User-Mode-Linux [Dike 2000]. As shown in this figure, there are four major modules in our prototype implementation.

— *Connection Tracker*. Who traces the incoming and outgoing 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 [Dike 2000].

5. EVALUATIONS

5.1. Linux/Slapper Test

Linux/Slapper [SLA] 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. It is different from many existing worms since it targets the buffer overflow in the heap of vulnerable Apache Web server 1.3 on Linux operating systems, including `RedHat`, `SuSe`, `Mandrake`, `Slackware`, and `Debian`.

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. The original source code of Slapper is 67655 bytes, and the uu-encoded source code is propagated between vulnerable hosts, which is of 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, 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 unrelated network addresses first as originally coded.

For the effectiveness experiments, the MAX_TIMEOUT is set as 2 minutes. 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 [King et al. 2003] has reported that compiling Linux 2.4.18 kernel inside UMLinux [Buchacker and Sieh 2001] 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 and disconnected the network at the worm's first exploit. Table I 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

Table I. Slapper: Infection and Code Transmission Time

	I_1		I_2	
	infection (s)	code TX(s)	infection (s)	code TX (s)
average value	9.3456	3.0654	91.8893	6.9773
standard deviation	0.4666	0.0120	1.2896	0.1103

Table II. 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	E-mail access
HTTPS	www.discovercard.com	E-transactions

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. 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.

To study whether we can detect worms in a mix of traffic, that is, 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 II are accessed repetitively from the host machine to test if WormTerminator would falsely take any traffic as worm traffic. Note that both interactive accesses such as e-mail accesses and E-transactions and noninteractive accesses are tested. 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 and disconnects the network.

5.2. WormTerminator Overhead

The implementation of WormTerminator inevitably introduces performance overhead to the protected system. This includes both the latency increase and the throughput reduction. As all traffic is directed to go through the virtual machine, all traffic is delayed. For requests waiting for being examined, the delay is dominated by I_2, and thus I_1, which varies depending on the worm itself. On the other hand, the throughput of the system will be affected since all TCP traffic has to go through the virtual machine.

To study the implementation overhead, we conduct experiments in the same local network with two machines. We measure the implementation overhead of splitter processing in the virtual machine, which is additional to the dynamically varying checking time, I_2, that an application has to wait for. Figure 4 illustrates our test setup. One of the machines is running an Apache 1.3.23 acting as a Web server, and the other is the host, where the virtual machine resides. From the host, a client sends out requests to access a Web page directly or to go through the virtual machine. The difference between them indicates the overhead imposed by WormTerminator.

To test the impact on latency, we have the client download a Web page of 1 byte. To test the impact on throughput, we have the client to download a Web page of 100 Mbytes. For each test, the client periodically sends requests to the server. The server

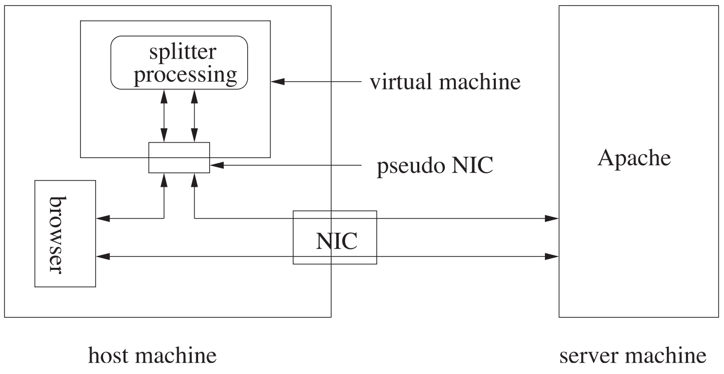

Fig. 4. Evaluation setup of WormTerminator implementation overhead.

Table III. WormTerminator Overhead: Latency and Throughput

	Latency (ms)		Throughput (MB/s)	
	average	std. dev.	average	std. dev.
direct Web access	0.681	0.0697	11.713351	0.0224
via WormTerminator	396.992	19.6012	2.287216	0.0251

Table IV. Virtual Machine Overhead: Latency and Throughput

	Latency (ms)		Throughput (MB/s)	
	average	std. dev.	average	std. dev.
direct Web access to the virtual machine	4.7423	0.0220	5.659181	0.0377

is warmed up with 10 requests for the page, so that the Web page to be accessed is cached in the memory of the server, thus avoiding any disk access uncertainty. The experiments have been conducted 100 times.

In the experiments, we used a text-based browser program. Its functions are to send out the requests, and wait for the reply from the server. After the requested document is received, the client completes its job. It does not involve the normal browser functions of displays and others, since they may disturb our measurements.

Table III shows the average result of 100 runs. Through WormTerminator, the delay is about 0.4 seconds, and the throughput is reduced to 20%. Compared to the direct access, such overhead is significant, although it barely affects a single user browsing experience as we have observed. The increased delay and reduced throughput may be due to our WormTerminator processing, or the virtual machine since WormTerminator heavily relies on the virtual machine, or both.

To further clarify the overhead sources, we first conduct the following experiments, in which the client program sends requests for 1-byte and 100-Mbyte Web pages to an Apache server running in a pure virtual machine.

Table IV shows the results over 100 runs. The result shows that the throughput of WormTerminator is roughly half of the virtual machine, which is roughly half of the direct host access. The 50% throughput reduction of WormTerminator (relative to a pure virtual machine) is because for the previous throughput tests, the reply message (100 Mbytes) goes through the pseudo network interface card (NIC) twice as indicated on Figure 4. These experimental results confirm that the throughput of our WormTerminator is largely limited by the virtual machine itself, particularly the pseudo NIC.

To dissect the latency caused by WormTerminator, we have the splitter with Squid2.4STABLE1 running in the host (a physical machine) directly. Then, the client

Table V. WormTerminator Real Processing Overhead: Latency and Throughput

	Latency (ms)		Throughput (MB/s)	
	average	std. dev.	average	std. dev.
splitter processing on the host	26.898	0.1200	11.674712	0.0235

Table VI. 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

program sends requests through it to access the Web server. Table V shows the average results over 100 runs. The results indicate that it only takes about 27 ms for the splitter processing on the host while it takes about 0.4 seconds in the virtual machine as shown in Table III. We thus believe such enlarged delay (about 15 times) is mainly due to the performance degradation of the virtual machine itself. After all, User-Mode-Linux itself is a user-mode process, which needs to wait for the context switch as other host processes, and single-file-based operations are slower than performed on a physical machine.

These sets of experiments show that the overhead of WormTerminator is dominated by the performance degradation of the virtual machine. As long as we improve the performance of the virtual machine, we can further reduce the overhead imposed by WormTerminator.

5.3. 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 previously, 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 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 VI briefly summarizes some statistics of client access logs.

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 5 shows the performance of the request cache when the cache size increases. The figure shows that when the cached objects are requested, 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.

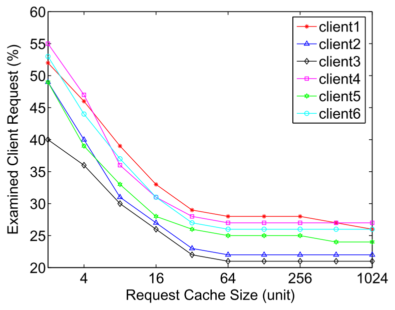

Fig. 5. Request cache effect: The portion of client requests that is affected. Note the x-axis is in log scale.

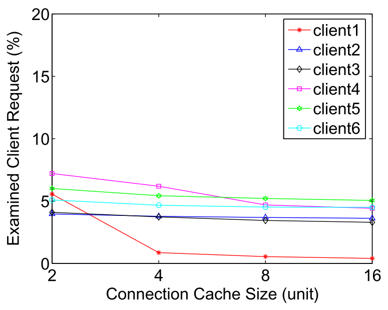

Fig. 6. Connection cache effect: The portion of client requests that is affected. Note the x-axis is in log scale.

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 6 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 expiration time.

6. DISCUSSION

6.1. Special Cases

As WormTerminator is designed to contain fast worms, it may not be effective to detect and contain slow worms. For example, a worm, after infecting some host, sleeps for a while before starting to infect other hosts. WormTerminator may fail to detect and contain such a worm. Nevertheless, WormTerminator could effectively contain most fast worms, which are considered more serious than slow worms.

If a worm is smart enough to check if the host where it has arrived is a virtual machine [Corey 2009; Hon 2004; Seifried 2002; Zou and Cunningham 2006], it may choose not to propagate. For example, Zou and Cunningham [2006] propose to exploit the legal and ethical constraints for honeypot detection, while software and hardware fingerprinting could also be generated [Corey 2009; Hon 2004; Seifried 2002]. In this case, WormTerminator will not be able to detect the worm traffic. This case, however, can be mitigated by instrumenting the physical machine to appear as a virtual machine. This would trick the smart worm not to propagate from the physical machine from the beginning. On the other hand, the research to anti-fingerprinting is also going on [NSF] as this is a challenge for all virtual machine-based solutions.

As discussed before, for some P2P applications, their traffic could be distinguished from the worm traffic by leveraging some application-level characteristics, including P2P queries and P2P downloading. On the other hand, if any worm propagates via P2P traffic, it may challenge the effectiveness of WormTerminator. Particularly, if the worm is stealthy and uses the P2P network for distributed coordination [Kataria et al. 2006], it would be very difficult to contain and stop such worms. A stealthy worm can wait long enough to pass the observation period set up according to our design, and then actively and quickly propagate like a fast worm. Although this stealthy worm does not match the definition of the fast worm (which should propagate as soon as it successfully infected one host) and has some delay between its infection and propagation, its threat is still severe. If the worm is synchronized using a distributed mechanism, once the worm on the host starts to propagate to other Internet hosts, WormTerminator should be able to observe similar behaviors on the virtual machine. Thus, correlation analysis between the host and the virtual machine could help distinguish such worm traffic from benign traffic. However, if the worm is not synchronized, the worm outbreaks on the host and on the virtual machine may be random and current design of WormTerminator cannot deal with such worms.

6.2. Other Limitations and Weakness

WormTerminator is a host-based approach. Fundamentally, a host-based solution demands a large scale deployment to be effective on the Internet. However, as a host-based approach, it complements existing network based approaches. That is, once some deployed hosts detect a fast spreading worm, they can report to existing network-based systems. Under such a situation, WormTerminators acts as “sensors” for network-based approaches.

In the current design of WormTerminator, we assume that a fast worm always targets a set of vulnerabilities. If a worm attack hosts with different vulnerabilities, a straightforward WormTerminator implementation may not be effective. However, if it also possible to set up multiple virtual machines to represent different vulnerable systems to accept diverted traffic to examine whether they could be infected. This, however, consumes more resources.

We also assume that the VMM is trusted. If the VMM is compromised and completely subverted by a worm, the worm can disable the VM, bypass the WormTerminator examination, etc. These can lead to the failure of worm detection. If an attacker is aware of the mechanism of WormTerminator, she can also bypass the examination by delaying its fast scanning. This, on one hand, evades WormTerminator detection. On the other hand, it also slows down the worm propagation. A careful setting of the threshold can greatly limit its propagation speed.

6.3. Implementation and Performance Optimizations

One of the key enabling techniques used in WormTerminator is the combination of application traffic replay and proxy. Since a worm may exploit the vulnerabilities of any normally used applications, the traffic replay needs to support all normal applications. One way to achieve this is to develop application-specific replay for each application protocol. In our WormTerminator prototype, we implemented the replay function for the normal HTTP and HTTPS requests. Ideally, we want some generic traffic replay functionality that can replay most application protocols. However, replaying application traffic often involves application-specific details (e.g., “cookie”), and it is quite difficult to make the traffic replay application independent. RolePlayer developed by Cui et al. [2006] has demonstrated the feasibility of such an application independent traffic replay. It has been shown that RolePlayer is able to successfully replay multistage infection process of the Blaster and W32.Randex.D worms based on at most two samples. Since the splitter can observe all the interactions between the traffic initiator at the host and the traffic responder at the VM, it can capture the complete traffic sample while the detector is examining the behavior of the VM. This would allow us to leverage the application independent traffic replay provided by RolePlayer in WormTerminator.

Besides the implementation obstacle, another important concern regarding WormTerminator is its overhead for normal applications, mainly dominated by I_2. We have also studied that the overhead for normal applications can be reduced with a cache if the application connection has been examined recently. We have also shown that our design naturally gets the support from the hardware technology development in multicore processors. Here we further discuss some other possible optimizations to further reduce the overhead from the following two aspects.

6.3.1. Improvement of Our Design. In its current design, after the traffic is diverted to the virtual machine for worm detection, WormTerminator keeps monitoring the network activities for a period upto T_{time} before making a decision. If the traffic is worm traffic, it is possible that the decision is made before T_{time} expires. However, if the traffic is normal traffic, WormTerminator always has to monitor for the entire T_{time} period. This directly translates to the delay to the application. In practice, such a decision could be made earlier for normal traffic. For example, if the diverted traffic is a HTTP request, and the object being requested does not exist in the virtual machine (which means it does not exist on the original host as well), it is very unlikely that this request is for worm propagation. Thus, once the server in the virtual machine replies, the traffic could be forwarded to the real destination immediately. Such an approach would be effective, but it is application dependent. That is, for different types of applications, the conditions may be very different. Thus, for a server supporting a particular service, such optimization would be efficient, while for a general client machine, such optimization could be difficult.

6.3.2. Improvement of Virtual Machine Performance. The performance of the virtual machine could be further improved. As WormTerminator relies on tracing to determine I_1, and uses I_1 and the slowdown SD of the virtual machine to determine I_2, the performance of the virtual machine is very important. While in the previous experiments we have found that the slowdown of User-Mode-Linux might be around 10 or more, a previous study [King et al. 2003] finds that compiling Linux 2.4.18 kernel inside VMware Workstation 3.1 2.4.18 kernel only takes 30% overhead relative to the directly compiling on the host OS. But compiling a kernel involves few network transmissions. In the worm propagation context, the network transmission delay must be considered. We have found in the experiments that transferring the worm code to User-Mode-Linux roughly takes a doubled time duration of transferring to a physical machine. We also have found that the pseudonetwork driver used in User-Mode-Linux impacts the system throughput. All these can be improved to further reduce the performance impact of WormTerminator. More efficient virtual machines could also be used. For example, VMware ESX server [Waldspurger 2002] and VMware Workstation [Sugerman et al. 2001] have been developed for x86 processors. Xen has also been widely studied and is being improved constantly [XEN a, b]. As our future work, we plan to study the performance of these alternatives and compare with that of User-Mode Linux. In addition, with the pervasive deployment of the dual-core and multicore systems, it is possible to utilize the additional core to dedicatedly run the virtual machine so that not only the host applications are not affected, but also the performance of the virtual machine could be improved.

7. CONCLUSION

Detecting and containing fast-spreading worms in real-time is very challenging, especially for those previously unknown or polymorphic worms. The key contribution of this article is that we have demonstrated that it is possible to detect and contain unknown, polymorphic worms in real-time while allowing all normal traffic to go out. Our worm detection and containment through WormTerminator are based on the defining characteristic of fast worms. WormTerminator detects the propagation of any fast worm before it can infect any other host on the Internet. This would allow us to contain 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. As the traffic diverting through the virtual machine increases the delay to normal traffic, we have also shown that applying caching can significantly reduce such overhead originated from the virtual machine.

ACKNOWLEDGMENTS

We would like to thank the reviewers for their helpful comments.

REFERENCES

Brumley, D., Newsome, J., Song, D., Wang, H., and Jha, S. 2006. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Buchacker, K. and Sieh, V. 2001. Framework for testing the fault-tolerance of systems including os and network aspects. In *Proceedings of the IEEE Symposium on High Assurance System Engineering (HASE)*. 95–105.

Corey, J. 2009 Advanced honeypot identification and exploitation. `http://www.phrack.org/fakes/p63/p63-0x09.txt`.

Cui, W., Paxson, V., Weaver, N., and Katz, R. 2006. Protocol-independent adaptive replay of application dialog. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS)*.

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

Hon. 2004. Honeyd security advisory 2004-001: Remonte detection via simple probe packet. `http://www.honeyd.org/adv.2004-01.asc`.

Kalla, R., Sinharoy, B., and Tendler, J. M. 2004. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro 24,* 2, 40–47.

Kataria, G., Anand, G., Araujo, R., Krishnan, R., and Perrig, A. 2006. A distributed stealthy coordination mechanism for worm synchronization. In *Proceedings of the 2nd International Conference on Security and Privacy in Communication Networks (SecureComm'06)*.

Kim, H. and Karp, B. 2004. Autograph: Toward automated distributed worm signature detection. In *Proceedings of USENIX Security*.

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

Kongetira, P., Aing-Aran, K., and Olukotun, K. 2005. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro 25,* 2.

Kreibich, C. and Crowcroft, J. 2003. Honeycomb - Creating intrusion detection signatures using honeypots. In *Proceedings of HotNets*.

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

Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., and Weaver, N. 2003. Inside the slammer worm. In *Proceedings of the IEEE Symposium on Security and Privacy*. Vol. 1.

NSF. Malware immunization through deterrence and diversion. `http://www.nsf.gov/awardsearch/showAward.do?AwardNumber=0650386`.

Paxson, V. 1999. Bro: A system for detecting network intruders in real time. *Comput. Netw. 31*.

Perdisci, R., Dagon, D., Lee, W., Fogla, P., and Sharif, M. 2006. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the IEEE Symposium on Security and Privacy*.

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

Seifried, K. 2002. Honeypotting with VMware basics. `http://www.seifried.org/security/index.php`.

Singh, S., Estan, C., Varghese, G., and Savage, S. 2003. The earlybird system for real-time detection of unknown worms. Tech. rep., University of California, San Diego.

Singh, S., Estan, C., Varghese, G., and Savage, S. 2004. Automated worm fingerprinting. In *Proceedings of OSDI*.

SLA. http://www.symantec.com/avcenter/venc/data/linux.slapper.worm.html.

Staniford, S. 2004. Containment of scanning worms in enterprise networks. *J. Comput. Secur*.

Sugerman, J., Venkitachalam, G., and Lim, B. 2001. Virtualizing I/O devices on VMware workstation's hosted virtual machine monitor. In *Proceedings of the USENIX Technical Conference*.

Waldspurger, C. 2002. Memory resource management in wmware ESX server. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

Weaver, N., Staniford, B., and Paxson, V. 2004. Very fast containment of scanning worms. In *Proceedings of USENIX Security*.

Williamson, M. 2002. Throttling viruses: Restricting propagation to defeat mobile malicious code. In *Proceedings of Annual Computer Security Applications Conference*.

XEN (a). http://www.cl.cam.ac.uk/research/srg/netos/xen/.

XEN (b). http://www.xensource.com/.

Zou, C. and Cunningham, R. 2006. Honeybot-aware advanced botnet construction and maintenance. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*.

Received March 2009; revised August 2009; accepted December 2009