An OS Independent Heuristics-based Worm-containment System

Uday Savagaonkar, Ravi Sahita, Gayathri Nagabhushan, Priya Rajagopal,
and David Durham

Abstract

We present an operating system independent and tamper-resistant worm-containment end-system. This system continuously observes outgoing network traffic over a finite-duration traffic window, and using heuristic rules executing in a secondary environment, detects infections. It automatically quarantines the infected host to stop further spread of the worm. We present four heuristic rules, and using network traffic traces collected from an enterprise network demonstrate that a port/protocol-tuned version of the heuristic provides lowest false-positives rate for different settings. Using simulations, we further evaluate the effectiveness of this heuristic in containing the spread of a worm in a medium-sized network. We then demonstrate that different window sizes are required for containing worms with different spread rates. Consequently, to be effective across a broad range of worms, we show it is advisable to use a heuristic that uses multiple window sizes. We also demonstrate that by effectively tuning the heuristic parameters and Dynamic Host Configuration Protocol (DHCP) server settings, one can contain worms with spread rates from 2 scans per second to upwards of 100,000 scans per second.

1. Introduction

Internet worms present a serious threat to today's highly networked computing environment. Unlike viruses and trojans, worms typically spread automatically without active human intervention, resulting in infection rates that are considerably higher than those of conventional viruses. For example, the Slammer worm attained probe rates of as high as 26,000 scans per second [14]. Additionally, the Witty worm outbreak demonstrated that patching may not be possible in time, given the worm appeared only one day after publication of the corresponding vulnerability. Thus, understanding the mechanism of automatic worm spread and developing appropriate containment strategies is important.

Modeling worm-spread is closely related with worm-containment, as worm-spread models allow us to analyze the effectiveness of a worm-containment strategy. Models proposed so far include differential-equation models [6, 8, 25] as well as Markov models [6, 11, 18]. Barring some disagreements regarding networks that demonstrate localized interactions [13], researchers agree that worms spread at exponential rates [8, 25] after initial infection. This exponential spread-pattern allows the network administrators extremely short reaction time to take any countermeasures [16].

The fast response times required emphasize the need for an automated mechanism to locally detect and control the spread of a worm. Traditionally, network administrators have used host-based firewalls and various intrusion-detection systems [10] for this purpose. Such systems attempt to prevent infection by scanning for worm signatures in the network traffic. Also, such firewalls prevent vulnerable services from being exposed to the network. Although these measures are effective against known worms, they are not effective against zero-day worm outbreaks. Moreover, most of these firewalls and intrusion-detection systems are software based, and hence are vulnerable to tampering and exploitation by worms themselves. Examples include the Witty worm [1] that infected a host intrusion detection software package.

Worms typically spread by exploiting some software vulnerability in the target system. For example, including the Morris worm [9] as well as the Code Red I and Code Red II worms [15], numerous worms have exploited different types of buffer overflow vulnerabilities [19]. Thus, researchers have proposed proactive mechanisms, such as, using robust-programming practices [7, 19], and generating automated tools that generate robust code [9]. However, these techniques require the existing software to be recompiled and/or rewritten. Also, such approaches mandate replacing entire software suite on the network, as any piece of software could expose vulnerability. Moreover, such mechanisms may result in performance degradation [9].

Recently researchers have proposed using the novel paradigm of worm containment—instead of preventing the infection, they detect the infection, and quarantine the infected host from the network [12, 16, 17, 21, 22, 23, 24]. Self-propagating worms spread by locating vulnerable hosts on the network, and compromising a vulnerable service running on those hosts [9, 15]. A typical mechanism used by worms for this purpose is called random address scan, where the worm randomly generates Internet Protocol (IP) addresses, and attempts to compromise vulnerable services on the hosts with those IP addresses. For example, the Code Red and the Slammer worms used random address scan to find vulnerable machines on the network. Other methods of scan, such as serial scan (where a worm scans IP address in a serial fashion), local preference scan (where a worm generates local IP addresses with higher probability), and divide-and-conquer scan (where a worm splits the range of IP addresses to scan when it propagates to a new computer) are also discussed in the literature [25]. Examples of worms using these other scanning techniques include Code Red II (local preference scan) and Blaster worm (serial scan) [25]. Worm-containment systems leverage this scanning behavior to detect infected hosts, and then, either throttle the traffic from the infected host [24, 17], or quarantine the infected host entirely [21, 17]. However, we note that, such systems, if implemented in OS-resident software, could be vulnerable to tamper and/or exploitation by worms. For example, nmap hackers were able to bypass the connection throttling mechanism implemented in the OS by sending out raw Ethernet frames onto the network [3].

Researchers have suggested increasing the effectiveness of such containment systems using Dynamic Host Configuration Protocol (DHCP) scattering to slow down the propagation of random-scanning worms [21]. In DHCP scattering, the network administrator uses IP addresses from a large private domain (such as, 10.10.0.0/16), and uses a Network Address Translation (NAT) device to map these IP addresses to external densely populated IP addresses. This address mapping makes random-scanning worms, including local preferential scan worms, encounter a large number of misses, resulting in a relatively slower spread of the worm. For example, if an enterprise with 1024 (2^{10}) computers uses a 10.10.0.0/16 address space, then the probability that any given random scan is successful is about 1/64, and thus the worm spends most of its time in the scanning phase.

In this paper we present a platform-based, OS-independent[1] worm-containment system. The system continuously observes outgoing network connections over a finite-duration traffic window (we refer to the duration of this window as the *window size*), and using heuristic rules, determines if the host is infected. For infected hosts, it automatically quarantines the host from the rest of the network to stop further spread of the worm into the network. We present four different heuristic rules that are extensions of the address-scan heuristic rule presented by Williamson [24]. We evaluate these four heuristic rules using 7981 hours of real enterprise network traffic, and demonstrate that a port/protocol-tuned version of the heuristic provides lowest false-positives rate for different window sizes. Using simulations, we further evaluate the effectiveness of this heuristic in containing the spread of a worm in a medium-sized network. Our simulations show that different window sizes are required for containing worms with different spread rates, and consequently, to be effective across a broad range of worms, it is advisable to use a multi-timescale heuristic (a heuristic that uses multiple window sizes). We also demonstrate that by appropriately tuning the heuristic parameters and using DHCP scattering, one can effectively contain extremely fast as well as extremely slow worms.

The rest of paper is organized as follows. In Section 2, we present related work and state our key contributions. In Section 3, we describe the system architecture, the various heuristics, and a prototype implementation. In Section 4, we provide simulation details and simulation results. In Section 5, we discuss the implications of our simulation results. We present our conclusions in Section 6.

[1] Our worm-containment system resides in isolated environment, and hence is robust against attacks on the OS.

2. Related Work and Our Key Contributions

Automatic worm containment is an active area of research [12, 17, 21, 22, 23, 24]. Most such systems assume that once a host is infected with a worm, it starts scanning the network for vulnerable hosts using random IP addresses. One of the best-known of such systems is the Threshold Random Walk (TRW) detection scheme proposed by Jung *et al.* [12]. They assume that, in normal behavior, a host generates more successful communication attempts than unsuccessful attempts and detect an infection using sequential hypothesis testing. The authors describe this system as an intrusion-detection system, even though it is straightforward to use it as a worm-containment system.

One of the short-comings of the TRW detection scheme is that typically it takes non-trivial time to determine whether a connection attempt is successful or not. Meanwhile, the worm could send out thousands of additional scan probes, infecting additional hosts before it gets quarantined. Schechter *et al*. [17] propose limiting the number of new first-time connection requests out of the host while outcome of previous requests is not known. However, they do not provide any analysis of what fraction of the network a worm would infect, if their system were deployed on every node in large networks. Note that they provide empirical analysis regarding what fraction of times they were able to detect a worm infection in real-life traffic traces. However, since at least a few worm-scan probes escape the containment system before detection, this ratio does not provide any insight into whether the worm would be stopped before infecting the entire network, if the worm were to attack a fully connected network (as is the case with most enterprise networks). The Weaver containment system [21] approximates the TRW using a *connection-cache*, which holds information about all the incoming and outgoing connection attempts. If the unsuccessful attempts exceed successful attempts by more than a pre-determined threshold, then the Weaver containment system reports an anomaly, and quarantines the host.

The above approaches are effective in catching traditional worms. However, if such systems were to become prevalent, future worms may specifically target these systems to circumvent their defenses. For example, a targeted worm can easily circumvent the above TRW-like systems [12, 17, 21], by continuously generating successful connections to known good addresses. In this manner, the difference between the number of successful connections and unsuccessful connection attempts will always be low, and hence the sequential hypothesis test will fail to detect the worm scan.

Whyte *et al*. [22] observe that normal network traffic results in Domain Name Service (DNS) lookups, while, address-scan traffic does not cause DNS lookups. Thus, they correlate the DNS activity with the outgoing connection activity to detect scanning activity. They present a similar technique correlating Address Resolution Protocol (ARP) traffic with outgoing traffic [23]. However, they acknowledge that this technique may not be applicable to all networking protocols, and may result in large number of false positives for various types of applications (e.g., peer-to-peer applications).

Williamson [24] leverages the observation that, over a small time period, an infected host makes a considerably larger number of connection attempts than an uninfected host. This is because, without the knowledge of the vulnerable hosts, the worm would have to attempt to infect many different hosts on the network in a short period of time to penetrate the network quickly. Williamson proposes delaying new outbound connections for containing such worms, but does not provide any insight into how effective this system is in containing worms. In other words, he does not provide any analysis showing by how much the worm would be slowed down, if his system were to be deployed in enterprise networks.

Here, we extend Williamson's address-scan detection scheme[2]. Our key contributions are as follows. We present an OS-independent architecture for the worm-containment system, and present four flavors of the address-scan heuristic. We use 7981 hours of real network traffic data from a corporate network to evaluate false-positive rate of these different flavors, and demonstrate that the port/protocol-based version performs the best at various values of record sizes (herein referred to as window sizes) and

[2] Our approach differs from Williamson's in that we quarantine the infected system entirely, instead of rate-limiting it. This, however, is acceptable, since as we show later, our false positives rates are extremely low.

threshold values. We use simulations to demonstrate the effectiveness of two such window sizes in containing worms with different scan rates under varying degree of DHCP address-space usage. Our simulations demonstrate that using multiple window sizes increases the effectiveness of the containment system.

3. Tamper Resistant Worm-containment System

3.1. System Architecture

3.1.1. Overview of the Architecture

Our worm-containment system works by applying heuristic rules to the outbound traffic from the host. If the heuristic rules detect an anomaly, the worm-containment system quarantines the host from the network. Figure 1 shows the architectural overview of our worm-containment system. As shown in the figure, our worm-containment system is divided into two units—the Inline Processing Unit (IPU), and the Sideband Processing Unit (SPU). Below we describe the functionality of each of these units.

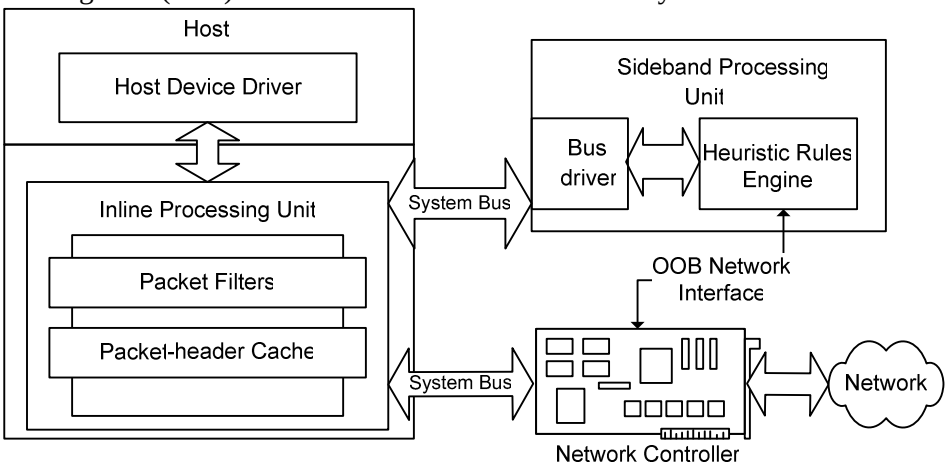

Figure 1: Tamper-resistant worm-containment architecture

3.1.2. Inline Processing Unit

The IPU includes the components that are in the direct path of the inbound/outbound host network traffic. Consequently, the operations performed by these components are bounded-time operations. There are two main IPU components—*Packet Filters* and *Packet Header Cache*. The Packet Filters operate on IP and TCP/UDP protocol header fields of the host network traffic. As a result of this filtering, the IPU may take specific actions, such as, dropping the packets. The Packet Header Cache holds a copy of time-stamped packet headers corresponding to inbound/outbound traffic. After passing through the packet filters, the packet headers are deposited into this cache. The SPU picks these headers up from this cache for sideband processing using standard platform buses (e.g., PCI [4]).

3.1.3. Sideband Processing Unit

The SPU includes the components that are not inline with the network traffic. These components analyze the traffic using the heuristic rules, and manage the configuration of the heuristics and Packet Filters in IPU. We delegate this functionality to SPU to avoid negative impact of heuristic processing on the system throughput. The SPU could be implemented in a tamper-resistant environment, isolated from the host operating environment. There are several possible ways of creating such a tamper-resistant partition. For example, the SPU could be implemented in a dedicated co-processor. Alternatively, the host could be partitioned using virtualization techniques, and the SPU could be implemented in a partition independent of the main operating system.

In our architecture, the SPU is endowed with an out-of-band (OOB) management channel. This channel is used to configure heuristic rules. Additionally, the SPU can use this communication channel to send out alerts and events to a remote administrator.

As shown in Figure 1, the SPU hosts the *Heuristic Rules Engine*, which applies the various worm containment heuristic rules (described later in this paper) to the packet headers read from the Packet Header Cache of the IPU. If the Heuristic Rules Engine detects evidence of anomalous traffic, it can install appropriate remedial Packet Filters into the IPU. Additionally, the Heuristic Rules Engine can send an alert to a remote administrator through the OOB management channel. Thus the inline and sideband processing elements constitute a closed-loop automated worm containment system.

3.2. Heuristic Rules

Our worm-containment system relies on heuristic rules to detect traffic anomalies that could indicate scanning activity by a worm. Here we present four such heuristic rules. These heuristic rules are devised based on the fundamental property of all self-propagating worms—a worm has to contact new hosts to spread into the network. Thus, all our heuristic rules identify events that suggest contact with a new host, and monitor such events for anomalous patterns.

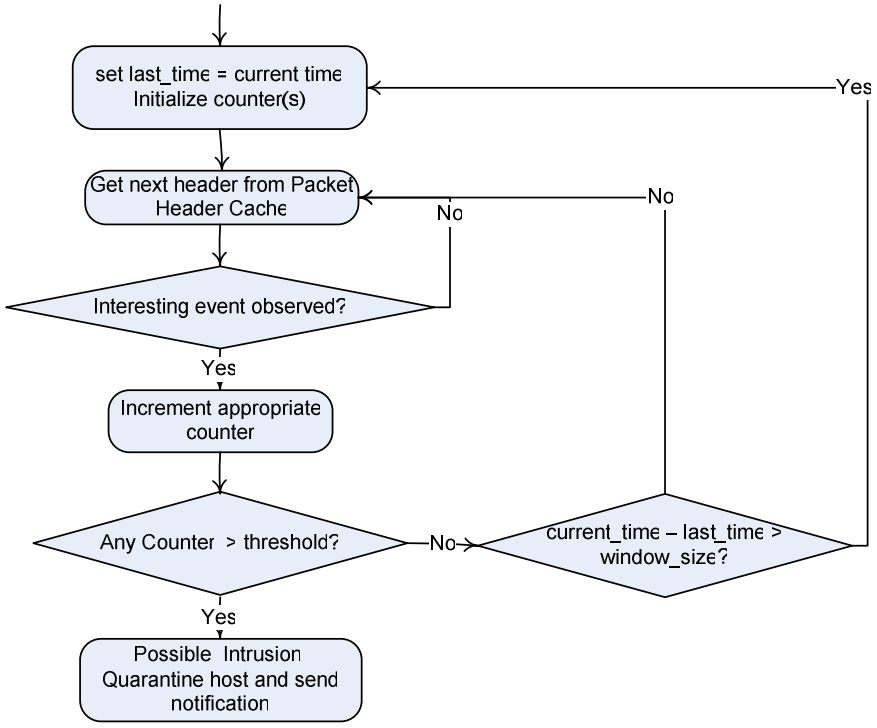

Figure 2: Basic operation of a heuristic

Figure 2 shows the basic mechanism employed by all of our heuristics. All of our heuristics observe packet headers, and count the number of "interesting events" in a given time interval. For this purpose, every heuristic exposes two configurable parameters—a window size and a threshold. The window size indicates the period at which a heuristic resets its counters. The threshold represents a limit value, which when crossed by the counter, indicates an anomalous event. As described earlier, if the heuristic detects an anomaly, the SPU quarantines the host from the rest of the network by installing appropriate packet filters in the IPU. Given this basic mechanism, the only aspect on which heuristics differ from each other is what they consider as interesting events and how they update their counters. Below we describe these four heuristics.

The New Connections Heuristic: This heuristic counts the number of unique connections observed by the worm-containment system in the given traffic window. A unique connection is described by a unique

triple, source port, destination port, and destination address. If the number of such connections in the traffic window exceeds the threshold value, the heuristic reports an anomaly.

The Pure Address Scan Heuristic: This heuristic counts the number of unique destination IP addresses in the given traffic window. If this number exceeds the threshold value, it reports an anomaly.

The Port-based Address Scan Heuristic: This heuristic counts the unique destination IP addresses grouped by destination port. If this number exceeds the threshold value for any destination port, it reports a traffic anomaly.

Port/Protocol-based Address scan: This heuristic divides all the traffic in the traffic window into TCP and UDP traffic. For the TCP traffic, it counts the number of unique destination IP addresses appearing in TCP-SYN requests grouped by destination port. For the UDP traffic, it counts the total number of unique destination IP addresses grouped by destination port. If any of these counters (either TCP or UDP for any port) exceeds the threshold value, it reports a traffic anomaly.

Note that all of these heuristics can be implemented as simple hash table lookups, followed by incrementing appropriate counters. Thus the heuristic rules engine can be implemented using a sideband processor with very low processing power.

3.3. Prototype Implementation

To test this architecture, we developed a system prototype using a commodity PC with a gigabit Ethernet network interface card running Linux OS (kernel version 2.4). We used a virtual machine implementation (Guest OS) on this platform to represent the infected-system. The IPU Packet Header Cache and the Packet Filters were implemented on the PC using Linux netfilter [2] and IPTables [5] respectively. The SPU was implemented on an Intel XScale® architecture-based IOP 80310 IO-controller card. The IOP80310 uses embedded Linux OS (kernel version 2.4.19). The heuristics engine was implemented as an application on the IOP 80310 board. The PCI channel was used for IPU-SPU communication. This communication interface is used by the Heuristics Rule Engine to periodically read the packet header cache as well as to configure the Packet Filters after traffic analysis.

We tested this prototype against four well-known, self-propagating worms—SQL Slammer, MSBlaster, CodeRed II and Slapper—by running their live copies on the virtual machine. Our prototype was able to detect all the four worms at reasonable parameter settings (explained later).

Note that the above IPU implementation is independent of the infected OS, as the infected OS is running on a virtual machine. Other OS-independent IPU implementation that we prototyped include

- Sampling the transmit ring buffers setup by the device driver from memory directly using the IOP80310 board. This implements the Packet Header Cache on the IOP80310 board.
- Implementing the IPU components on the IOP80310 board and funneling traffic through the IOP80310 board. This implements both the IPU components on the embedded processor.

We mention here that when we sampled the device driver ring buffer using the IOP80310, we were able to capture only 3 to 5 percent of the traffic headers. Alternatively, the IPU can be implemented in hardware as part of communications controller as shown in Figure 1.

3.4. Performance Analysis

We analytically demonstrate that the above worm-containment system is not CPU intensive. We assume that the SPU is implemented using an embedded 100 MHz processor with clocks-per-instruction of 1, cache line width of 64-bytes, and memory latency (to fetch one cache line) of 400nS (these are reasonable assumptions for low-end commodity memory and embedded processors). For our heuristics, the only fields of interest are the destination address, destination port, protocol, and TCP flags. The packet header cache can store these fields from one header using a 16-byte block. Thus, on average, it takes the SPU 100nS to fetch each header from the packet header cache. Noting that all of our heuristic algorithms can be implemented as simple hash look-ups, it is safe to assume that our heuristic takes no more than 50

instructions[3] (500nS) to process one packet header from the packet header cache, if it does not encounter a cache miss. Allowing additional 50 clock cycles (500nS) for cache misses, and embedded OS operations, we need about 1100nS to process each header. This implies we can process over 900,000 headers per second. Now, assuming that an average packet size of 200 bytes[4], we expect a 1Gigabit Network Interface Card (NIC) to deposit at most 1e9/(8*200) = 625,000 headers in the packet header cache. Thus, we have enough processing power to support a Gbps NIC operating at its full speed. In reality, we expect the number of headers in the packet header cache to be much smaller than the estimated in this analysis because of various platform latencies, framing overheads and non-TCP/non-UDP traffic. It should however be noted that this analysis only deals with IPv4 traffic, and IPv6 traffic would result in different numbers.

4. Simulations

4.1. Data Collection and Simulation Setup

We evaluate our worm containment system using two metrics—the false positives rate and the effectiveness of the system in stopping various worms. A false positive is an event when the worm-containment system determines the host to be infected, even when the host is not infected. The other metric, effectiveness of the system in stopping various worms, represents whether the system is capable of stopping a worm, and what percentage of the network the worm infects before all infected hosts are completely quarantined. We use simulations to evaluate our system using these two metrics.

Our simulations consist of two parts. The first part—called Sim1—simulates the behavior of the worm-containment system presented above on traffic traces collected using windump or tcpdump. We use this simulator to measure the false-positive rate of the various heuristics presented above on real network traffic collected from variety of client host systems located in a corporate network of a large corporation. Overall, we collected IP network usage traces on 39 client host systems. The client host systems were chosen from a cross section of the corporation, including manufacturing, information, technology, and networking labs. These client systems were distributed geographically around the world, however, majority of them were located in the United States of America. We collected the data traces over a period of about 40 days. At the end of the data-collection effort, it was determined from system logs that one of the client systems could have been infected during the data collection epoch, and hence traffic traces from this system were not used for analysis. The total length of our network traces (excluding the potentially infected system) was 7981 hours. The information technology system logs indicate that these network traces represent worm-free traffic, and hence we use these traces with Sim1 for false-positives analysis.

The second simulator, called Sim2, simulates a medium-sized (8192 hosts) computer network under a worm attack. We assume that every host on the network is reachable from every other host, and that these hosts are uniformly spread across a large DHCP address space (size of the DHCP address space is a simulation parameter). We assume that all of hosts on the network are vulnerable to the worm attack, and stay connected for the entire duration of simulation. We assume that the network has enough bandwidth so that the worm-spread traffic combined with the legitimate traffic does not cause any congestion (ideal conditions for worm spread).

For Sim2, we model our network as a Markov chain, and simulate the worm propagation using transition probabilities of the Markov chain. In this aspect, our simulations resemble the work of Spears *et al*. [18] and Billings *et al*. [6]. In our simulations, each host has one of three states—susceptible, infected, and quarantined. We start our simulations in a state where ten randomly selected hosts are in the infected state, and all the other hosts are in the susceptible state. We simulate the worm propagation as a discrete-time Markov chain, with the time epoch set to 100 microseconds. The maximum duration of a simulation

[3] Wang [20] presents numerous hashing algorithms that can be implemented using less than 20 instructions.
[4] This assumption is reasonable, as the headers from various layers add up to more than 50 bytes, and worms need to add some payload to spread. For example, the Slammer worm used UDP packets that were 404 bytes in size [14].

run is 5 hours (1.8e+8 epochs), as we believe that this is sufficient time for the network administrator to take effective countermeasures against the worm spread. At each epoch, each infected host generates a random number of scan probes to scan for susceptible hosts on the network. The number of scan probes generated by a host is a Poisson random variable with mean as determined by the worm spread rate, and the IP address for each probe is chosen uniformly from the entire DHCP address space (note that the DHCP address space, typically, has more addresses than the number of hosts on the network). If an IP address for the scan probe matches the IP address of a susceptible host, the susceptible host transitions to infected state. Each of the outgoing probes from an infected host is observed by the worm-containment system on that host, and the system quarantines the host when the heuristic rule reports a traffic anomaly. A quarantined host does not scan for additional vulnerable hosts.

Thus, each run of Sim2 presents us a worm-propagation trajectory, describing the number of machines in each state (susceptible, infected, and quarantined) at each time epoch. Averaging the number of infected and quarantined machines across multiple trajectories provides us with average worm penetration (expressed in number of hosts) into the network at each time epoch. Note that our Sim2 simulations do not take into account any background, legitimate traffic. In reality, such traffic would also be observed by the worm-containment system, and hence the real worm-containment system may trigger sooner than the simulated worm-containment system. This would result in even faster quarantine of the infected host. Thus, in one way, our Sim2 results represent the worst-case performance of the worm-containment system.

4.2. Simulation Results

To evaluate the false positives rates of the four heuristics (New Connection, Pure Address Scan, Port-based Address Scan, and Port/Protocol-based Address Scan), we applied these heuristics to the 7981 hours of enterprise network traffic traces using the simulator Sim1. For these evaluations, we used window sizes of 1mS, 5mS, 50mS, 1S, and 50S. For each window size, each heuristic was evaluated with a threshold value of 2, 4, 8, 16, 32, 64, 128, and 256. Table 1 shows the number of false positives per hour of network traffic produced by each of these heuristics. It can be seen from Table 1 that the Port/Protocol-based Address Scan heuristic gives the lowest false positives for any window-size/threshold combination. Consequently, this heuristic attains zero false positives for the smallest threshold value among all heuristics for any given window size. Thus, this heuristic can be used with most stringent threshold values across all window sizes keeping the false alarm rate zero (or extremely low).

In Table 1, we also estimate the worms each heuristic would catch at various values of window size, if it were to use the smallest threshold value that gives zero false positives. These estimates are based on the spread rates of five well-known worms—Slammer, Code Red II, Slapper, and MSBlaster, as reported on various Internet web sites (e.g., [14]). Each cell with zero false positives includes letters representing the worms that would be stopped with the setting in that cell. The convention used is a-Slammer, b-Witty, c-Code Red II, d-Slapper, and e-MSBlaster.

Next we evaluate how effective our worm-containment system is in stopping the worm using Sim2. For demonstration purpose, we use window sizes of 1mS and 50S (smallest and largest from Sim1). Also, we use the Port/Protocol-based Address Scan heuristic—because of its superior false-positives performance—for our Sim2 results. For the window size of 1mS, we choose the threshold value of 8, and for the window size of 50S, we choose the threshold value of 64. These are the smallest threshold values for which this heuristic yields zero false positives on our data set. This way, we demonstrate the effectiveness of our worm-containment system at reasonable parameter settings.

Table 1: The number of false positives per hour produced by each of the four heurstics at various values of the window size and the threshold parameters. The table also estimates the various worms that would be caught by various settings. See Page 8 for more explanation.

New Connections Heuristic

Threshold	Window Size: 1mS	5mS	50mS	1S	50S
2	105.47	189.88	523.27	1680.2	7671.9
4	33.187	59.784	162.56	760.54	4605.3
8	2.9664	9.9703	35.732	280.25	2334.3
16	0.2778	1.2394	6.5588	69.342	903.07
32	0.0038	0.1227	0.3327	10.377	248.92
64	0	0_{ab}	0.0033	0.4077	70.785
128	0	0	0_{ab}	0.0435	14.399
256	0	0	0_{ab}	0_{ab}	0.3355

Pure Address Scan Heuristic

Threshold	Window Size: 1mS	5mS	50mS	1S	50S
2	85.219	129	263.41	961.35	6612.2
4	28.38	47.296	78.414	257.25	3400.5
8	1.365	6.3988	15.978	49.312	803.98
16	0_a	0.0243	0.095	0.8211	84.43
32	0	0_{ab}	0.0154	0.092	3.3634
64	0	0_{ab}	0_{ab}	0.0048	0.6677
128	0	0	0_{ab}	0_{ab}	0_{abcde}
256	0	0	0_{ab}	0_{ab}	0_{abcd}

Port-based Address Scan Heuristic

Threshold	Window Size: 1mS	5mS	50mS	1S	50S
2	31.304	32.764	45.505	91.099	490.6
4	16.339	16.792	17.308	20.929	86.588
8	0.0332	0.1083	0.149	0.349	12.705
16	0_a	0.0229	0.0685	0.1004	1.2372
32	0	0_{ab}	0.014	0.0318	0.2209
64	0	0_{ab}	0_{ab}	0_{ab}	0_{abcde}
128	0	0	0_{ab}	0_{ab}	0_{abcde}
256	0	0	0_{ab}	0_{ab}	0_{abcd}

Port/Protocol-based Address Scan Heuristic

Threshold	Window Size: 1mS	5mS	50mS	1S	50S
2	0.7116	1.671	5.9771	38.852	396.77
4	0.0853	0.1296	0.6069	4.3086	53.749
8	0_{ab}	0_{ab}	0.0001	0.1965	12.578
16	0_a	0_{ab}	0_{ab}	0.0003	1.175
32	0	0_{ab}	0_{ab}	0_{abcd}	0.1812
64	0	0_{ab}	0_{ab}	0_{ab}	0_{abcde}
128	0	0	0_{ab}	0_{ab}	0_{abcde}
256	0	0	0_{ab}	0_{ab}	0_{abcd}

Figure 3 shows the network penetration of worms with various spread rates as a function of DHCP address *prefix size* for the two parameter settings above (window size = 1mS, threshold = 8, and window size = 50S, threshold = 64). Each point on these graphs was obtained by averaging ten trajectories of the worm-spread generated using Sim2 (different random-number generator seed for each trajectory). The *X* axis in these graphs shows the prefix-size of the DHCP address space. A prefix-size of *n* implies that all the addresses in the DHCP address space have first *n* bits in common. Thus, for example, the DHCP address space with a prefix-size of 8 has first 8 bits fixed, and consequently has 2^{24} addresses in it. As the prefix size increases, the address space has fewer addresses available in it. Thus, in these graphs, as the DHCP prefix size increases, the DHCP usage becomes increasingly dense. This is because the total number of hosts in the system is constant, however with increasing DHCP prefix size, available number of addresses in the address space decreases. The *Y* axis in these graphs shows the level of worm penetration in 5 hours in terms of percent vulnerable hosts infected.

As can be seen from these figures, the heuristic with window size of 1mS and threshold of 8 is capable of stopping only the fastest-spreading worm, the one with a scan rate of 100,000 connections per second. On the other hand, with sufficiently small DHCP prefix size, the heuristic with window size of 50S and threshold of 64, is capable of stopping all the worms, except for the slowest worm (with a scan rate of 0.5 connections per second). This certainly is not a surprise, considering that any worm with a spread rate of less than 8000 scans per second is expected to fly under the radar of the heuristic with window size of 1mS and threshold of 8. However, it should be noted that, for the fastest spreading worm (100,000 scans per second), this heuristic configuration is capable of stopping the worm with much larger DHCP prefix sizes than the 50S/64 connections configuration. This is because the 1mS/8 connections configuration is capable of detecting the worm scan much earlier (only after about 8 scan probes) than the other configuration (after about 64 scan probes), and hence is more effective in stopping the worm.

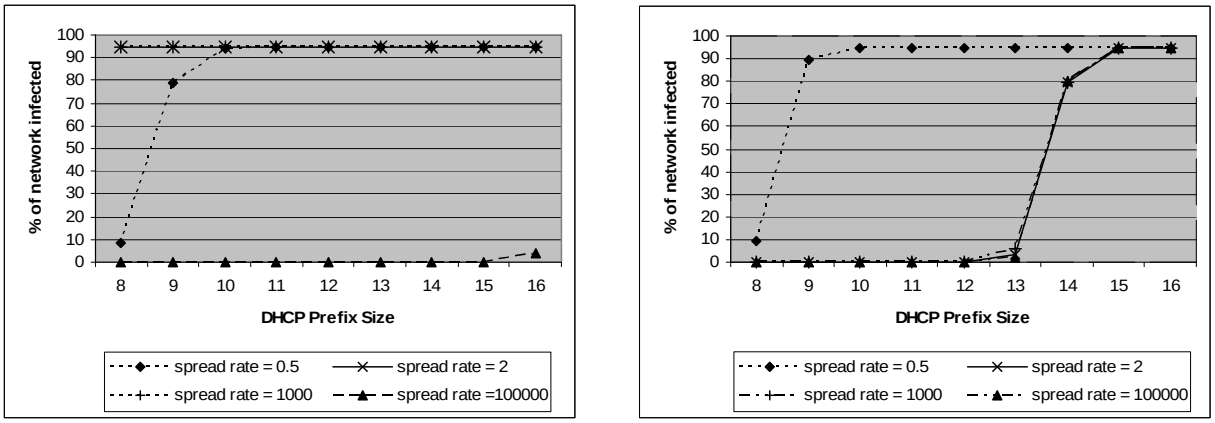

Figure 3: Worm-penetration as a function of DHCP address-space size (a) Effectiveness of 1mS window with threshold set to 8, (b) Effectiveness of 50S window with threshold set to 64

This implies that a combined heuristic, which would use both, the 1mS window and the 50S window, with appropriate threshold values, would be effective across a wider range of worms. Also, as we have chosen the threshold values to give zero (or very low) false positives, the resulting combination would also result in zero (or very low) false positives.

Also, note that even though the worm with spread rate of 0.5 connections per second spreads to the entire network for almost all network prefix sizes, the speed of penetration of such a slow worm is very slow. Sim2 can be used to look at these numbers. Along with the maximum network penetration attained, each run of Sim2 also provides time it takes for the worm to attain this level network penetration for that run (this could be less than 100%). We note down these times for each of the 10 random runs used for plotting Figure 3, and consider the minimum of the 10 time values for each setting. This value gives us the estimate of how much time, at a minimum, the worm would take to attain its maximum level of penetration into the network. Figure 4 plots these time values as a function of the DHCP prefix size. As can be seen Figure 4, for both the heuristic settings, the slow worm (0.5 scans per second) worm takes over 5000 seconds (about 1.5 hours) to penetrate networks with prefix sizes of 11 and below. We believe that this gives ample opportunity for the network administrators to react to such worm outbreaks.

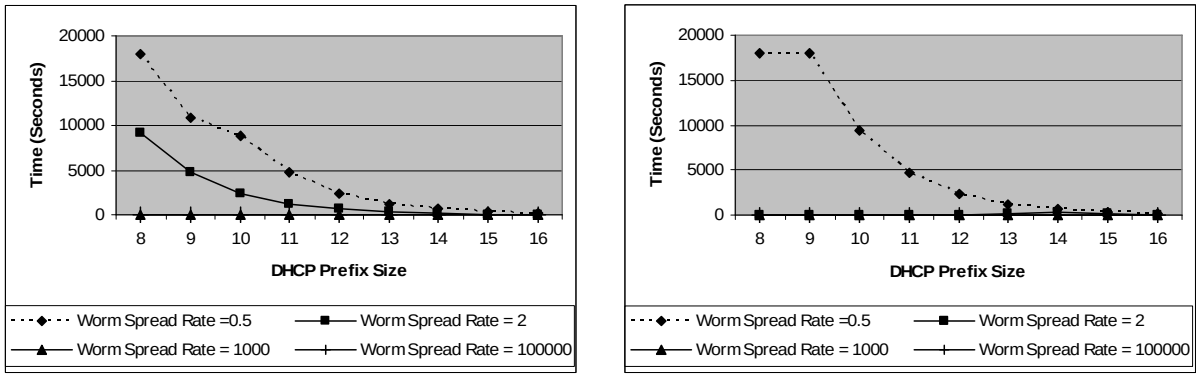

Figure 4: Minimum time for worm to attain its maximum penetration as a function of DHCP address-space size (a) Time taken with 1mS window with threshold set to 8, (b) Time taken with 50S window with threshold set to 64

5. Discussion

We demonstrate that multi-time scale heuristics provide a better defense against a wide range of worms. Our conclusion to employ multi-timescale heuristic is based on the results shown below. Using

multi-timescale heuristic, one can monitor worms with a large range of propagation speeds. This can be achieved without generating false positives by tuning the threshold value at each timescale to achieve zero false positives at that timescale. Multi-timescale heuristics clearly have an advantage over single-timescale heuristics based worm-containment systems. Table 2 summarizes the effect of the two window sizes on worms with various spread rates from our Sim2 runs. The table lists the worm penetration into the network for the lower window (1mS—L in the round braces), as well as the upper window (50S—U in round braces). As can be seen, the heuristic with the smaller window size is more effective for the fast worm, where as the heuristic with the larger window size is effective for slower worms. However, the threshold values for both of these window sizes were chosen to give zero false positives using the Sim1 results. Thus, our results demonstrate that the multi-timescale heuristics are capable of intercepting a range of slow as well as fast spreading worms, without a prohibitive number of false positives.

Table 2: Effectiveness of different timescales in stopping the worm before it infects the entire network against different worm-propagation rates

		Address Space Size (Prefix Size)		
		Small (14-16)	Medium (11-13)	Large (8-10)
Worm Speed	Fast (100000 scans per second)	80-100% (U) 0.15-3% (L)	0.2-2.2% (U) 0.13 (L)	0.13-0.14% (U) 0.12 (L)
	Medium (1000 scans per second)	80-100% (U) 100% (L)	0.16-5% (U) 100% (L)	0.12-0.14% (U) 100% (L)
	Slow (2 scans per second)	80-100% (U) 100% (L)	0.16-3% (U) 100% (L)	0.12-0.14% (U) 100% (L)
	Very Slow (0.5 scans per second)	100% (U,L)	100% (U,L)	8-100% (U) 100% (L)

Our simulation and prototype results show the effect of modifying network parameters, such as DHCP prefix sizes (DHCP usage), on the spread of self-propagating worms. Patching of systems is a slow process, and therefore DHCP address space control provides a better tool for such control. We also note that, in our experience, DHCP usage in enterprise networks is typically clustered in address subspaces. Such allocation is beneficial to worms that have simple spread address generation methods, such as linear address scan or subnet preference scan. Thus, we recommend randomly distributing the allocated IP addresses uniformly across the DHCP address space. We also recommend using extremely large private DHCP addresses, and then using a global NAT device (one that keeps a 1:1 mapping between external and internal IP addresses) for exposing the hosts to the Internet.

Our multi-timescale heuristics still suffer from some deficiencies. As can be seen from Table 2, very slow worms could penetrate a large portion of the network largely undetected. Also worms that employ target lists could use a divide and conquer scanning approach [25] to achieve fast network penetration even at slow propagation speeds. However, note that our architecture has an OOB channel from the SPU to other entities on the network. This channel can be used to communicate the local triggers of the worm-containment system to the backend processing unit, which can use centralized inference mechanisms for detecting worms [21].

6. Conclusions

We demonstrate a tamper-resistant, OS independent approach to zero-day worm detection and containment. We have tested a prototype of this system using four flavors of the address-scan heuristic against live, well-known worms, and have verified the capability of the prototype to contain these worms. We simulate the effect of our worm-containment system on real enterprise traffic to demonstrate that, with appropriate values of heuristic parameters, it is possible to achieve extremely low false-positives rate. Using our own network simulator, we also demonstrate that these parameter settings can effectively contain worms with various spread rates. Our simulations indicate that different parameter settings are more effective against different worm-spread patterns. Thus, we propose the use of multi-time scale

heuristics to provide coverage for the range of slow and fast propagating worms. We demonstrate the effect of DHCP address prefix size on the spread of self-propagating worms. Large address spaces prefixes possible using IPv6 or NAT devices can take advantage of this concept to reduce the effectiveness of self-propagating worms. Also, an administrator can estimate the risk profile of their networks for newer worms as they are discovered using the analysis approach provided in this paper.

References

[1] —. Symantec Report – Witty Worm. Available from http://securityresponse.symantec.com/avcenter/venc/data/w32.witty.worm.html

[2] —. Net filter public repository. http://www.netfilter.org.

[3] —. Nmap Development: Potential Windows SP2 fix http://seclists.org/lists/nmap-dev/2004/Jul-Sep/0030.html

[4] —. Peripheral Connect Interface specification. *http://www.pcisig.org*

[5] O. Andreasson. IPTables Tutorial 1.1.19. http://iptables-tutorial.frozentux.net/iptables-tutorial.html

[6] L. Billings, W. M. Spears, and I. B. Schwartz (2002). A Unified Prediction of Computer Virus Spread in Connected Networks. *Physics Letters A, 297, 261-266.*

[7] M. Bishop. Robust Programming. Technical report. University of California, Davis, California, 2002. http://nob.cs.ucdavis.edu/bishop/secprog/robust.html.

[8] Z. Chen, L. Gao, and K. Kwiat. Modeling the Spread of Active Worms. In Proceedings of the 22nd Annual Joint Conference of the IEEE Computers and Communications Societies (INFOCOM), volume 3, pages 1890-1900, San Francisco, California, April 2003.

[9] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. *In Proceedings of the 7th USENIX Security Conference, pages 63-78, San Antonio, Texas, January 1998.*

[10] D. Denning (1987).An Intrusion Detection Model. IEEE Transactions on Software Engineering, 13(2):222—232

[11] M. Garetto, W. Gong, and D. Towsley. Modeling Malware Spreading Dynamics. In Proceedings of the 22nd Annual Joint Conference of the IEEE Computers and Communications Societies (INFOCOM), volume 3, pages 1869-1879, San Francisco, California, April 2003.

[12] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. *In Proceedings of the 2004 IEEE Symposium on Security and Privacy, to appear, 2004.*

[13] J. Kephart and S. White. Measuring and Modeling Computer Virus Prevalence. In Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy, pages 2-14, Oakland, California, May 1993.

[14] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The Spread of the Sapphire/Slammer Worm. Technical Report, The Cooperative Association for Internet Data Analysis (CAIDA). http://www.caida.org/outreach/papers/2003/sapphire/sapphire.html.

[15] D. Moore and C. Shannon. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In Proceedings of the 2002 ACM SIGCOMM Internet Measurement Workshop, pages 273-284, Marseille, France, November 2002.

[16] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirement for Containing Self-propagating Code. In Proceedings of the 22nd Annual Joint Conference of the IEEE Computers and Communications Societies (INFOCOM), volume 3, pages 1901-1910, San Francisco, California, April 2003.

[17] S. E. Schechter. J. Jung, A. W. Berger. Fast Detection of Scanning Worm Infections. In Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID), French Riviera, France, September 2004.

[18] W. M. Spears, L. Billings, and I. B. Schwartz. Modeling Viral Epidemiology in Connected Networks. *NRL Memorandum Report NRL/MR/6700--01-8537.*

[19] J. Viega and G. McGraw. Buffer overflows. In Building Secure Software: How to Avoid Security Problems the Right Way, Chapter 7, pages 135-185. *Addison-Wesley, Boston, Massachusetts, September 2001.*

[20] T. Wang. Integer Hash Function. Technical Report. Avalable from http://www.concentric.net/~Ttwang/tech/inthash.htm.

[21] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In Proceedings of the 13th USENIX Security Symposium, pages 29-44, San Diego, California August 2004.

[22] D. Whyte, E. Kranakis, and P. van Oorschot. DNS-based Detection of Scanning Worms in an Enterprise Network. *In Proceedings of the 12th Network and Distributed System Security Symposium (NDSS), pages 181-195, February 2005.*

[23] D. Whyte, E. Kranakis, and P. C. van Oorschot. ARP-based Detection of Scanning Worms in an Enterprise Network. *Technical Report, School of Computer Science, Carleton University, October 2004.*

[24] M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. In Proceedings of the 18th Annual Computer Security Applications Conference (ACSAC 2002), December 9–13, 2002

[25] C. Zou, D. Towsley, and W.Gong. On the Performance of Internet Worm Scanning Strategies. On the performance of lnternet worm scanning strategies, Umass ECE Technical Report TR-03-CSE-07, 2003.9:1-15. Submitted to Journal of Performance Evaluation.