

WORM vs. WORM: Preliminary Study of an Active Counter-Attack Mechanism

Frank Castañeda*†
castanef@us.ibm.com

Emre Can Sezer†
ecsezer@ncsu.edu

Jun Xu†
jxu3@ncsu.edu

Pervasive Computing Division*
IBM Software Group

Department of Computer Science†
North Carolina State University

ABSTRACT

Self-propagating computer worms have been terrorizing the Internet for the last several years. With the increasing density, inter-connectivity and bandwidth of the Internet combined with security measures that inadequately scale, worms will continue to plague the Internet community. Existing anti-virus and intrusion detection systems are clearly inadequate to defend against many recent fast-spreading worms. In this paper we explore an active counter-attack method - anti-worms. We propose a method that transforms a malicious worm into an anti-worm which disinfects its original. The method is evaluated using the CodeRed, Blaster and Slammer worms. We show through simulation the effectiveness of an anti-worm with several propagation schemes and its impact on the overall network. We also discuss important limitations of the proposed method.

Categories and Subject Descriptors

D.4.6 [Operating Systems]: Security and Protection—*invasive software (e.g., viruses, worms, Trojan horses)*; C.2.0 [Computer-Communication Networks]: Security and Protection—*worms*; K.6.5 [Management of Computing and Information Systems]: Security and Protection—*invasive software (e.g., viruses, worms, Trojan horses)*

General Terms

Experimentation, Security

Keywords

Anti-worm, Good worm, Worm

1. INTRODUCTION

We consider computer worms that do not require human activation. Such a worm infects a server by exploiting a vulnerable application, usually through a specially crafted TCP or UDP message. It then continues to infect other servers

with the same vulnerability. The questions we explore in this paper are: how to deploy an active immunization mechanism and how effective is it?

Current defense mechanisms such as anti-virus (AV) software and Intrusion Detection Systems (IDS) passively analyze file system activities or network traffic. Both systems use pre-defined attack signatures provided by AV/IDS vendors for detection. Vendors use network traffic monitors and analysis tools to discover new attacks and update their signature database. Users then need to download the signatures once they are updated. There may be significant delays at both the vendor and end-user signature updates before new attacks will be recognized. Staniford et. al. [28] predicts that, with better scanning algorithms, it is possible for worms to infect 90 percent of the susceptible hosts in mere minutes. Recent fast spreading worms (e.g., Slammer, Blaster) have proven this, and these worms clearly defeat current defenses. Anomaly detection systems use statistical methods to detect attacks but have not been very successful due to high false positive rates [1, 7].

Another defense is to ensure that each system is up-to-date with latest vendor patches. Many system administrators are reluctant to apply patches until they have been thoroughly tested in their environment. Combined with the magnitude of patches being released, an organization can be easily overwhelmed with patch testing and software updating. Small businesses and end consumers simply cannot keep up with the pace. It has been observed that most worms are actually created after the exploited vulnerability has been published on the Internet [6]. Our study of the Symantec worm and virus database [31] also shows that many recent worms are discovered weeks after Microsoft had released the patches. This clearly shows that manually applied patches are not effective in countering worms because they require human reactions and they are usually slow and do not scale well. Automatic patching at the software vendor level has also been considered, but end users basically do not feel comfortable with software vendors pushing new code to their systems. Not only does the consumer lack the ability to test the code, but there now exists a new security problem: how much do you trust your vendor [5, 6].

Due to the inadequacies in current defense mechanisms, it is imperative that better defenses be developed to detect a worm and to immunize hosts before the worm reaches epidemic proportions. Weaver et. al. [34] have also pointed out the need for better defense mechanisms and the possible research areas in this field. In this paper we explore an active immunization method by the use of a *good worm* or *anti-*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WORM'04, October 29, 2004, Washington, DC, USA.

Copyright 2004 ACM 1-58113-970-5/04/0010 ...\$5.00.

worm. The idea is to transform a malicious worm into an anti-worm which spreads itself using the same mechanism as the original worm and immunizes a host.

We propose an architecture to generate an anti-worm. There are several major challenges in the architecture: new worm detection, transforming a worm into an anti-worm, and anti-worm propagation schemes. New worm detection is an ongoing research area. Designing and implementing a completely automated anti-worm generator may require extensive research. We leverage a Honey-pot based system [24] in our current proposal for worm detection and capture. The focus of this paper is on worm transformation and propagation schemes. After we capture a new worm we transform the message of the original worm into an anti-worm through a payload search algorithm, we then generate anti-worm code and embed it in the payload. The job of the anti-worm code is to stop the activities of an existing worm in a system, and detect and prevent new attacks against the hosting machine (i.e., *immunization*). The transformation method is evaluated using three recent worms: CodeRed, Slammer and Blaster. An important requirement for an anti-worm is that it must be able to suppress the overall activities of the original worm, and in the process, not pose significant traffic load on the network. We show the effectiveness of four anti-worm propagation schemes (passive, active, hybrid, and IDS-based) through simulation.

The paper is organized as follows: §2 highlight the unresolved issues and limitations of the proposed approach; §3 discusses background and related work in active defense mechanisms; §4 discusses the anti-worm generation and transformation algorithms; §5 presents the anti-worm propagation schemes and simulation results; we conclude with §6. We discuss related work throughout the paper.

2. ISSUES AND LIMITATIONS

Later in the paper, we propose four types of anti-worm propagation methods: a passively spreading anti-worm which responds to worm probes with a counterattack; an active-scanning anti-worm which actively finds targets and propagate to them; a hybrid scheme that combines the features of the passive and active anti-worms; and an IDS based anti-worm where an IDS or similar device intercept attack messages and counter-attacks [21]. However, the proposed anti-worms in their current forms have several unresolved issues and limitations that are almost certain to cause serious deployment problems. We highlight some of these issues in this section.

Patching Worms. A patching worm can be in various forms: explicit or implicit. A worm could patch a system explicitly by removing the vulnerability that it exploits in the system. *Welchia* [15] is such an example, downloading the patch to prevent other worms from competing with it. Patching occurs implicitly if the worm infects a single-threaded service, where the act of infection blocks further exploitation. Common worms such as Slammer [18], Witty [29], and (to a limited extent) Blaster [30]

¹ all infected such services and blocked secondary infection

¹Blaster blocked the RPC service during infection until the next reset. This block was only temporary, however, as Windows XP automatically resets after 1 minute of RPC failure, while Windows 2K is commonly reset by users as RPC fail-

attempts. A patching worm will completely disable the proposed passive anti-worm in its current form. The effect of the active-scanning is limited to only protecting uninfected systems, while IDS-based strike-back can only replace an infectious attempt monitored by an IDS, therefore limiting its effectiveness.

Network Impact. The main goal of the proposed anti-worm is to stop the spreading of malicious worms. For this purpose, the anti-worm itself needs to be generated quickly and spread at least as fast as the original worm. There could be several problems with this. A worm could spread so fast that either the traffic due to a very fast anti-worm could cause similar damage, or the anti-worm would be too slow to stop the original worm. In the case of the Slammer worm, even if a perfect anti-Slammer can be generated automatically, it would be at least as disruptive as the original Slammer for the duration of its spread.

In addition, our method relies on the original worm to generate the corresponding anti-worm. A buggy worm may result in a buggy anti-worm, which may lead to instability in the target network. Even without bugs, the active anti-worm is equally disruptive to the network during the spreading process, although this disruption may be more transitory if it is programmed to stop after a certain period of time.

Legal Issues. It is currently grossly illegal to intrude a system that one does not control. Legal and trust issues make it hard for anti-worms to be accepted as an active defense mechanism. The current proposed Internet-scale anti-worm will also have legal issues across judiciary boundaries, where applicable laws may vary in different countries. Cross-jurisdictional issues could stop governments, as well as private entities, from employing anti-worms. This limits the use of anti-worm to local confined domains. For example, a private company can choose to use anti-worm within its corporate network where it has full legal access. Research has shown [27] that limiting the spread of a worm, even ones with manual designers, is often very hard.

Network and Patch Management. Current or future enhanced network management tools and patch management tools can accomplish similar goals without aforementioned technical problems even if the anti-worms can be limited to systems fully-controlled by a defender. A lot of the patch management infrastructure is not in place yet, but the current systems can be further enhanced. If one has administrative control over a set of machines, these methods offer advantages over the proposed anti-worm schemes. For example, proposed patch-management systems could allow you to preventively immunize systems in a controlled domain within seconds, faster than scanning worms can propagate. Likewise, infected machines can be constrained or removed by network management, by techniques such as VLANing suspicious systems onto a different virtual network, isolating them from the rest of the enterprise.

We have discussed only some of the major limitations of our approach. The proposed anti-worm in its current form will have practical usage problems. In spite of these

ure stops both cut and paste, as well as drag and drop from working.

probably serious limitations, we still find the idea of an automatically generated anti-worm an intriguing one. The techniques developed here (transforming a worm into an anti-worm with its own payload, and anti-worm propagation schemes in particular) would certainly be interesting to other researchers for studying future worms and for inventing new techniques.

3. BACKGROUND

Anti-worms pose several novel features, for one they allow for a reactive measure to malicious worms that can potentially be deployed with no additional infrastructure in place. Anti-virus software, firewalls, IDS and IPS are generally static in nature and only prevent known attacks, therefore must be updated periodically to stay current. We have already shown that humans do not scale against fast spreading worms. We also find that many computer systems today still have major security holes [25].

Automatic patching systems present a security risk since the vendor now has full control over what software is running on your computer. Secondly, these patching systems are only effective against worms that exploit known vulnerabilities that have patches available. Anti-worms, however, do not need to carry a patch. They may block traffic to the affected service with application specific filtering and detection. Anti-worms can also notify users of specific worms and vulnerabilities to reduce the risk of incompatible and un-tested patches.

To date, anti-worms and their effect have not been fully explored. We discuss some existing worms that we consider anti-worms below.

The **Welchia** (Anti)Worm is a Blaster worm variant. This worm exploits the issue that was addressed by Microsoft Security Bulletin MS03-026, a buffer overflow vulnerability in the Microsoft Remote Procedure Call (RPC) service. The Welchia and Blaster worms both use TCP port 135 to spread. The Welchia anti-worm uses the same vulnerability to infect hosts, it then downloads the MS03-026 patch and reboots the system to immunize the host. The combined effect of both worms was devastating to the network, because of the bandwidth used to download the patch by Welchia and the DoS attack on windowsupdate.com by Blaster. One variant of Welchia attempted to win over the Blaster worm by using 300 propagation threads instead of the original 50, theoretically giving the anti-worm a 6x scanning rate. Welchia was more destabilizing than blaster, because it used an unrestricted ICMP scanner with a short timeout, which generated considerably more network traffic. This is what destabilized the navy marine corps intranet. There were mixed reactions towards the anti-worm [15]. Whether Welchia had a "good" intent is questionable, as at least some Welchia variants reportedly contained code to create an unrestricted back-door on all infected systems. Rather, Welchia's patching and blaster-removal functions can also be interpreted as the worm author's attempt to remove competitors from the Internet ecology, and as a tool to prevent duplicate infections without the complexity or possibility of errors in other techniques, such as the one used by the Morris worm [4].

The **CRClean** Anti-Worm is a Code-Red II variant. This worm exploits Microsoft Security Bulletin MS01-033, a buffer overflow vulnerability in the Index Server plug-in for Mi-

crosoft's web server. Though the CRClean worm was never released it presented some novel ideas; first of all it was a passive worm, meaning it only spread to machines that attempted to attack it. This "counter-attack" scheme prevents the network perturbation normally caused by actively scanning worms. Secondly, it would add a filter to intercept the Code-Red attack before it infected the machine, rather than attempt to download the patch from Microsoft, thus having a "blocking" effect against future infections. In this paper, CRClean is an excellent example of the schemes we plan to investigate: block and counter-attack [13].

The **Code-Green** Anti-Worm is another Code-Red II worm variant which contains a payload designed to remove Code-Red and download the MS01-033 patch from Microsoft and install it. This worm, like CRClean was never released on the Internet [8].

The **Cheese** Anti-Worm spreads through a backdoor opened by the Lion Linux worm. The Lion worm uses a buffer overflow vulnerability in the Transaction Signature (TSIG) part of a DNS/BIND message on UDP port 53. The Cheese Anti-Worm removes the backdoor created by Lion and the vulnerable service from inetd. Both the worm and anti-worm were released on the Internet but were very sparse and there was little known effect on the network or system resources. This may be due to the wide knowledge of the exploit and small number of susceptible hosts [26].

4. ANTI-WORM GENERATION

We first describe a general framework for detecting, capturing, and analyzing an Internet worm, and for generating its anti-worm. Figure 1 illustrates the proposed architecture. The framework is divided into three stages: (1) worm detection and capture, (2) worm analysis, and (3) anti-worm generation. We investigate the feasibility of stages (1) and (2) and suggest some potential ideas for further research. The focus of this paper, however, is on stage (3). We implement our proposal for anti-worm generation and have evaluated the algorithm using three real Internet worms: Code Red I, MSBlaster, and SQL Slammer.

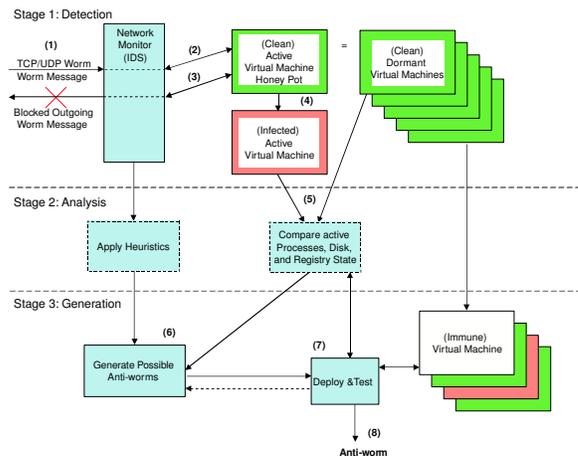


Figure 1: Architecture

Stage 1 - Worm Detection and Capture. The challenge for this stage is to be able to distinguish between net-

work messages from worm traffic and non-worm traffic. Existing work on *Honey Pot* based IDS systems [24] may be leveraged for this stage. A honeypot is a monitored system connected to a network as a decoy, whose sole purpose is to collect traffic from scanning worms and attackers. By definition, a honeypot system has no legitimate traffic. The system is only accessible to an entity that is performing random scanning. The system uses a database of signatures of known traffic which is ignored. This can be any traffic that is seen before and has already been analyzed and recognized. The worm detection and capture process starts as soon as any new message enters a honeypot system. The honeypot can use LCS (Longest Common Substring) based pattern matching on messages entering and exiting the system until a certain threshold of similar messages are seen on the network before signaling a worm presence. A vertical and horizontal LCS scheme was used by Kreibich et. al. [14] when creating IDS signatures using honeypots, and has proven to be successful against Code-Red and Slammer worms.

At Step (1) in Figure 1, the worm message enters the system and the network monitor records the message. At this point the system does not know if the message is a worm or not. In (2) the message is received by an active Virtual Machine (VM) image which is running the application suite containing a vulnerable application. The use of VMs is based on the assumption that the vulnerable application is unknown and not easily discernible. In (3) and (4) the now infected VM retransmits the same or similar (replayed) worm message. Here the network monitor waits for this repeat message (with an alternate destination) to determine that VM is infected and attempting to spread the worm to others. The source and destination port number of the attack message and the message itself is recorded and sent to the generation module.

Stage 2 - Worm Analysis. At Step (5) in Stage 2, the VM may be compared with a dormant version of the VM. The dormant VM is one that was created from a backup of the now infected VM some time before it was infected (there may be several backups created from different times.) In (6) data is collected from the comparison of the before-infection and after-infection VMs, including what files changed on the system, what registry entries were changed and what processes were started. This data may then be used to generate an anti-worm that repairs any registry changes or file changes or processes started by the malicious worm.

Stage 3 - Anti-Worm Generation. This stage is the main focus of this paper. The goal of the anti-worm generation algorithm is to produce network messages that use the same attacking method used by the original worm messages, and to generate executable code used by the anti-worm to disinfect compromised systems. We propose a method that accomplishes the above goal. The proposed method is logically divided into two steps: (1) transform the original worm messages into anti-worm messages; and (2) generate the executable code used by the anti-worm. The next two subsections describe these two steps in more detail.

4.1 Transforming a Worm into an Anti-Worm

Given the network messages used by the original worm, we want to transform them into ones that will embody the entire anti-worm. We want to retain the part of the original worm message that is used to compromise a remote machine with a certain vulnerability. The reason for the retention is because we want the anti-worm to use the same attacking method used by the original worm. While retaining the attack vector, we want to overwrite the part of the original message that is the malicious executable. Anti-worm code generation is discussed in §4.2.

For example, the SQL Slammer Worm (shown in Figure 2) uses a stack buffer overflow vulnerability to compromise a remote server, and then propagates itself to infect more vulnerable hosts. Our method retains part of the Slammer worm that overruns the remote unchecked buffer, finds the beginning of worm executable payload, and finally overwrites the executable payload using the generated anti-worm code.

We define a worm message to be the TCP or UDP data which consists of the exploit vector and the payload. We define the worm *payload* as the portion of a worm messages that consists of the worm executable code and the corresponding data used by this code. We define the *exploit vector* to be the portion of the data which facilitates the execution of the payload. In other words, the exploit vector contains the data necessary for the worm to compromise a susceptible application and transfers control to the malicious payload code. In the case of the SQL Slammer worm, the exploit vector is the part of the message that is used to overrun the unchecked stack buffer in SQL server which includes some dummy data bytes and most importantly the return (jump) address that points to the malicious payload. Figure 2 illustrates the message structure of the SQL Slammer worm.

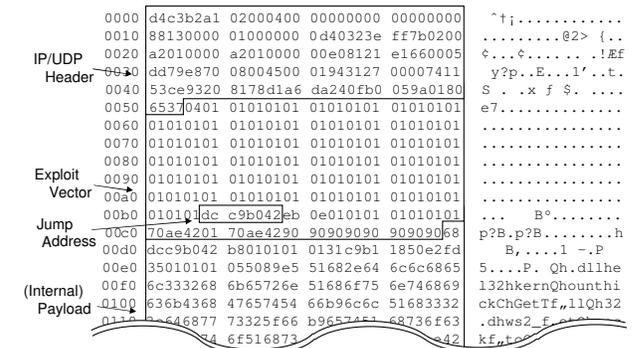


Figure 2: SQL Slammer Worm Dissected

A worm may also contain an *external payload*, that is, a payload downloaded in an *out-of-band* channel by the payload embedded in the worm message. An example of an external payload can be found in the MSBlaster worm. It uses a TFTP channel installed by MSBlaster on the infected host to transfer itself to other hosts.

Our transformation method takes as input the original worm message, and produces as output the corresponding anti-worm message. Our method must find address α - the address value which points to the end of the worm's *exploit vector*, or equivalently, the start of the worm's *payload* in the

worm message. For a new worm that has not been seen before, we do not know the value of α and must find its correct value. The search for the correct α value is implemented using a trial-and-error approach. The algorithm starts by setting α to 0 (or a better value based on some heuristics, e.g., ignore the network and transport layer headers). The algorithm then copies the anti-worm code to the address range $[\alpha, \alpha + S]$ in the original worm message, where S is size of the anti-worm code. The transformed message is then sent to a Virtual Machine (VM) running a vulnerable server for testing. If the anti-worm message causes the server to crash or some other unexpected behavior, the current value of α is not a correct guess. The algorithm increments α and repeats the above steps until the generated message achieves the goals of the anti-worm. This algorithm ensures that the malicious payload is overwritten with the anti-worm payload, and meanwhile, retains the original exploit vector.

The time it takes to complete this process is a critical factor in the ability to combat an active and fast spreading worm. On a modern 2GHz machine, the test process takes approximately 30 seconds per iteration. The process involves an OS image with saved state using Microsoft Virtual PC 2004. The image is loaded from a saved state, thus bypassing the boot-up stage. The worm is then transmitted to the VM image and verified. Afterwards the VM is terminated without saving any disk or system state, the old system saved state file is restored from a backup and the VM is restored (The VM's XML based configuration files facilitates this automation.) For SQL Slammer worm ($\alpha \approx 100$), the search without any heuristics takes about 50 minutes in serial. For larger worms such as MSBlaster, it will take a lot longer without. However, the trial-and-error search algorithm for α can be greatly improved using heuristics and parallelism. We have used several heuristics in our experiments (details in §4.3): ignoring protocol headers, finding NOP bytes, and considering application protocols. The search algorithm can also be fully parallelized using a large number of virtual machines since there is no dependency between iterations.

4.2 Generating Anti-Worm Code

The anti-worm code must perform a number of tasks in order to be useful. It must stop the original worm from causing further damage to the infected host; it must stop the original worm from infecting and propagating to other hosts; it should clean up the damage caused by the original worm on the infected host; and it must be able to disinfect other infected or susceptible hosts. We first discuss the implementation of our current prototype and its strategies, we then present how the current implementation may be improved.

4.2.1 Current Prototype

As a proof-of-concept, we have implemented a prototype anti-worm generation algorithm on the Microsoft Win32 platform. The prototype includes two parts: the anti-worm bootstrap code, and the external payload code. The bootstrap code is the payload of the anti-worm message. It downloads the external payload from a well known site and launches it, and then terminates the vulnerable server process². The external payload immediately binds to and listens on the network port that is used by the vulnerable

²Simply terminating some system services may destabilize

application. Upon detecting incoming messages that match the worm signature, the external payload spreads itself to the origin of these messages for further immunization. The prototype is intended to demonstrate the feasibility of our ideas. In particular, we want to show that the search algorithms presented in the previous section are practical, and we are able to stop the malicious worm and disinfect the compromised machines.

A crucial factor in any worm is the ability to call into various system APIs necessary for successful propagation. Our first prototype hard-coded the addresses of the required APIs in the anti-worm code. This failed because these addresses change based on minor releases, language releases, and even the order various system libraries are loaded in. For this reason, we employ a mechanism to find these function addresses on the fly by locating the function addresses in the executable's Relative to Virtual Address (RVA) table based on the known address of a single DLL/EXE [23].

In Windows, the base address of an executable (all Windows executables are in PE format) is the address where the PE header is loaded. This base address may vary. Sample code available from RootKit.com [9] assumes that the executable is always loaded at 0x0040,0000 (the default base virtual address). However, applications may choose to use a different base address. For example, during our experimentation with Microsoft IIS, we found that IIS is actually loaded at base address 0x0100,0000. We modified the RootKit code to calculate the base address based on the runtime load address of kernel32.dll. The system library kernel32.dll is loaded by every windows application at or around 0x77E6,0000, our anti-worm bootstrap attempts to locate this dll between 0x77E0, 0000 and 0x7800,0000.

In our experimentation, we found that many worms exploit C-string based buffer overflow where a key requirement is that the messages do not have a Null terminating string. It is very difficult to guarantee this condition for the generated anti-worm. Instead, we used an obfuscation technique where most of the anti-worm is XOR'ed with a pre-determined value (X), except for the initial part of the anti-worm. When the anti-worm begins to execute, it first XORs X with the body of the anti-worm. The value of X is calculated when the anti-worm code is generated to guarantee that no Null characters appear in the code.

In the current prototype, the bootstrap code is generic and can be used by different worms. It is implemented in approximately 400 lines of hand-coded x86 assembly. The object code is about 700 bytes. The bootstrap code is not optimized and we believe its size can be substantially reduced. The external payload is written in C. It has a generic part that can be shared by all worms. In addition, each anti-worm needs to embed their own worm signature for runtime detection.

4.2.2 Improving the Prototype

The current prototype stops the active worm by killing the infected application process. This serves to inform the system's owner and to protect the system from future worm attacks. The denial of service effect caused by this may be rectified by creating a smarter anti-worm capable of changing the host address and/or port binding of the affected application and redirecting only normal traffic to the new bind-

the system (e.g., RPC on Windows), making this simple approach disruptive in some cases

ing. We call this an application-specific intrusion detection system. The system intercepts and inspects all messages sent to the application. Messages that match the signature of the current worm are not forwarded to the vulnerable application.

The anti-worm code must also send itself to the originators of attacking worms for immunization and disinfection. Our current prototype uses an external payload for this purpose. While simple, it has a serious drawback: The extra traffic generated by the downloading of the payload may lead to network congestion. It may also lead to denial of service at the download site. Downloading the payload can always be done through a Content Delivery Network (e.g., Akami) which can reduce greatly the network traffic. A better way is to have the anti-worm re-generate itself. The anti-worm residing on a host has the complete information (code and data) as an entity; it may be possible for it to assemble network messages that are an exact replication of itself. The re-generation algorithm is quite sophisticated and requires further investigation.

4.3 Experimentation with Real Worms

Our process is evaluated through tests against the following well studied worms: Code-Red I, MSBlaster, and SQL Slammer. The OS platform used in our experiments is the Microsoft Windows 2000 Server with Service Pack 2, Microsoft SQL Server 2000, and Microsoft Internet Information Server (IIS)³. Tools were developed to simplify the experiments but some tasks were still performed manually (e.g., restarting and reloading of the virtual machines). More work will be required to completely automate the process.

Code Red I. Code-Red utilizes a buffer overflow in the indexing IIS plug-in, `idq.dll`. The worm utilizes an HTTP post request for `default.ida` containing a query parameter designed to overflow the buffer in this plug-in and overwrite the stack return address. The body of the post contains the worm's payload, which is allocated and stored in the heap and can be of any practical size. Because of this the Code-Red worm message may contain the entire worm malcode and there is no need for an external payload. The worm message causes the return instruction pointer (EIP) to be overwritten with `0x7801,CBD3` which points to a piece of code in `msvcrt.dll`, a critical dll to the function of IIS. This code when disassembled is `"call ebx"`, where EBX at this point contains a pointer to the posted data (the worm's payload) [22].

The length of the payload is 3,569 bytes, and the exploit vector is 470 bytes, totaling 4,039 for the entire message. The worm is transmitted in a single TCP flow. A simple heuristic based algorithm can easily guess the location to start replacing the worm code with the anti-worm code to be at α equal to 470. Since the protocol is HTTP and a post request is being made the generation may assume that the payload falls after the end of the HTTP header, which is always ended with `0x0D0A,0D0A`. This case will yield a successful anti-worm in a single test. In the case where the generation is not capable of using such a heuristic the worm generation would require 470 trials, if using a sequential search from $\alpha = 0$.

³Microsoft Windows 2000, Microsoft SQL Server, and Microsoft Internet Information Server are registered trademarks of the Microsoft Corporation

MSBlaster. MSBlaster utilizes a buffer overflow in the Distributed Common Object Model (DCOM) Remote Procedure Call (RPC) service, which is always enabled by default [30]. This vulnerability was addressed by Microsoft Security Bulletin MS03-026 shortly after the vulnerability was discovered by LSD [16]. This vulnerability was widespread since many systems were not firewall protected and the common use of RPC by many applications.

The DCOM exploit consists of two TCP messages with destination port 135. The first message, 72 bytes, is a RPC bind request which contains no malcode, it simply set up the system for the second request. The second TCP message, 1,704 bytes, consists of RPC request containing the buffer overflow intended to write a jump address pointing to a call which executes the malcode now stored in the heap, similar to Code-Red's exploit. α is somewhere within 908 and 1,036 of the second message. A heuristic capable of locating this value can look for a series of "nop" (0x90, no operation) instructions commonly found prior to the exploit code, used as padding for jump offsets that may not always start execution at the same place. Secondly the heuristic can also search for other sequences of what would be useless single instructions like "inc eax" (0x40, increment a register.) The list of useless instructions is limited to only a small set of single byte instructions. Single byte instructions are required since multi-byte instructions will not be interpreted correctly if the instruction stream is not executed starting from the first byte of the instruction.

SQL Slammer. The MS-SQL Worm, also known as the Sapphire or Slammer Worm, was the fastest spreading worm ever released. The worm uses a vulnerability reported in Microsoft Security Bulletin MS02-039 to target the Microsoft SQL Server Resolution Service which listens on UDP port 1434. This service is used to manage multiple instances of the SQL Server but is always enabled even in single instance configurations. The stack based buffer overflow is caused by a UDP packet starting with a 0x04 byte which tells the service to resolve a name. This name is str-copied into a stack allocated buffer of 16 bytes and not checked for length. The result is a stack based buffer overflow in the `strcpy()` function in `msvcrt.dll` [3]. Its small size (a single UDP packet total 404 bytes) and simple transport mechanism combined made it one of the fastest spreading worms ever. The worm was released at 05:30 UTC on January 25, 2002, and it is estimated that 90% of the susceptible hosts (~75,000) were infected in a little more than 10 minutes [18].

The small size of Slammer posed an interesting challenge. Our prototype anti-worm code is about 700 bytes, which is far larger than Slammer's 404-byte single UDP packet. The α value for this worm is between 117 and 171. A second constraint is that the exploit vector and payload of the anti-worm must fit within a 1,000 byte buffer used by the `recv()` socket call in the target application. Currently, we append the 700 bytes of the anti-worm code beginning at α , making the resulting UDP packet twice as large as the original one. While this works for Slammer, it clearly shows that an anti-worm should be no larger than the original worm in order to achieve its goals. This is one of the major reasons that we split our anti-worm into two parts, the bootstrap and the external payload.

Other Worms. We have yet to experiment with multi-vector worms such as Nimda. In these cases, an anti-worm would need to be created for each attack vector. The propagation payload of the anti-worm can be extended to contain multiple vectors in a single payload, in order to bridge networks with mixed vulnerabilities. Otherwise, if the propagation payload contained only a single vector per vector detected at the IDS level, crossing into firewall protected networks may be more difficult for our anti-worm as opposed to a hand crafted worm.

Worms with variable messages (not re-playable messages), for example a worm that needs to negotiate a connection or change the jump address of the attack vector (i.e., the address is computed from values only known during the network exchange,) currently will not produce successful anti-worms. More investigation is required to determine how these complex network interactions may be captured. One possibility may involve simulating the worm in a sand-box environment and learning the protocol from the simulated worm. Meanwhile, current-day worms do not appear to be using such sophisticated network communications, and as these communications become more and more complex the slower the scanning rate of the worm and the smaller the worm threat becomes.

5. PROPAGATION SCHEMES

Once an anti-worm is developed, immunization becomes a race against the malicious worm. Ideally, we would want the anti-worm to spread to all vulnerable hosts as soon as possible, not only preventing the malicious worm from compromising other hosts but also working against it in an effort to rid the network of the worm threat. We have shown that with adequate resources, an anti-worm can be developed in a reasonable amount of time once a worm has been detected. In this section, we discuss several anti-worm propagation schemes and evaluate the schemes using simulation:

- A *Passive Anti-Worm* listens on a host and waits for attacks from the original worm. It spreads itself (i.e., counter-attacks) to an attacking worm only. The propagation scheme minimizes additional network traffic. The CRClean anti-worm uses this mode.
- An *Active Scanning Anti-Worm* randomly scans the available IP address space and transfers itself to those compromised or vulnerable hosts for immunization and disinfection. This scheme can quickly counter the original worm attack but may incur large amount of extra scanning and message traffic and cause undesirable network congestion.
- An *Active-Passive Hybrid Anti-Worm* combines the features of the two schemes above. It starts as an active scanning worm initially and switches to passive mode when sufficient number of hosts have been installed with the anti-worm.
- An *IDS-based Anti-Worm* involves the use of dedicated intrusion detection sensors throughout the Internet, scanning suspected packets and identifying the sender and receiver for immunization. This scheme was proposed and simulated in [20, 24].

In the remainder of this section, we first describe our simulation methods, and then present simulation results for the anti-worm propagation schemes.

5.1 Simulation Method

We first discuss some related work in worm propagation simulation and then present our method. In the early 90's, scientists from IBM adapted mathematical models of epidemiology to computer viruses [10, 11, 12]. In a simple epidemic model, a host is either infected (I) or susceptible (S). Once a host is infected, it is assumed that it stays in that state. This model is used in a number of research areas, usually slightly modified to simulate different phenomenon [33, 32, 28]. In the General Epidemic Model [35], a host is in infected, susceptible or removed state. Removal can be interpreted as immunizing the host or blocking it from the network. In either case, the host cannot be reinfected.

Recent studies have used the spread of Code-Red as a comparison basis for their simulations [36, 28, 2]. In the epidemic models, parameters such as the pairwise infection rate are often adjusted to fit the data. Zou et. al. [37] have studied the effect of dynamic quarantine of hosts on worm propagation. Moore et. al. [19] have studied two methods for containing malicious code and their effect on worm propagation: address blacklisting and content filtering.

We have developed an iterative simulation that allows us to simulate different scanning modes for both the malicious worm and the anti-worm. Our simulation parameters include: the percentage of online hosts, the number of the susceptible hosts, the TCP timeout value, the number of threads the worm runs on a host, and the percentage of infected susceptible hosts when the anti-worm becomes available. From the first three arguments we calculate the average scanning rate for the worm. The simulator tracks the number of infected and cured hosts.

At every iteration, for each infected host, a random host is chosen from the available IP address space, and a random number is generated to determine the state of the target host. A host can be in one of the following states: offline (OL), infected (I), cured (C), susceptible (S) or not-susceptible (NS) state. An iteration is a single scan by every infected host. A *second* in simulation time consists of *scan rate* number of iterations. After each iteration, the infected, susceptible and cured host numbers are updated, along with the probabilities associated with them. If the anti-worm spreading scheme scans IP addresses as well, then the same steps are applied to every cured host too. Our simulator uses the complete (2^{32}) IPv4 address space.

To realistically evaluate the different anti-worm propagation strategies, we use Code-Red I version 2 (CR-Iv2) as the base worm for simulation. CR-Iv2 runs 100 threads, 99 of them performing random IP scanning using TCP with a time out value of 21 seconds. Assuming that only 25% of the IP address space is online (similar percentage is used in [32]), we calculated the average scanning rate of a host to be around 6 scans per second. This complies with reports that approximate this value to be between 5 and 11. We used an approximated vulnerable host number of 360,000, the estimated number of vulnerable hosts for Code-Red I [17].

Figure 3 shows our simulation results for the propagation of Code Red Iv2 starting with a single infected host. The figure shows that the Code Red Iv2 spread in about 10 hours to all vulnerable hosts without any immunization. Network measurement in Moore [17] shows the real worm reach full coverage in about 14 hours. However, many systems were patched during the measurement period and hence slowed

down the worm propagation. In fact, the exponential growth phase in our simulation, 6-9 hours after initial infection, matches that of the real data in [17]. Since our goal is to evaluate anti-worm propagation, we believe that our simulation can serve well for the purpose.

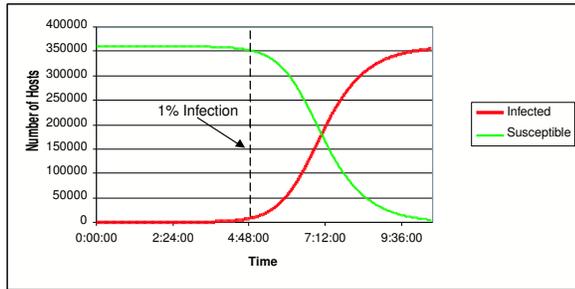


Figure 3: Code Red Iv2 Propagation Starting with a Single Infected Host

It takes time for an anti-worm to be developed, by which time the malicious worm must have infected part of the susceptible hosts. In simulating the effect of anti-worms, we model this initial delay using the percent of infected hosts at the time of anti-worm release. We use a value of 1% for Code Red Iv2, i.e., 3,600 hosts being infected when the anti-worm is released. Using results from Figure 3, it means that an anti-worm is developed within 5.5 hours of the malicious worm release (the vertical 1% line in Figure 3). Note also that all the anti-worm simulations assumes non-patching behavior on the part of the malicious worm.

5.2 Passive Anti-worm

A passive anti-worm will listen on the susceptible port for an infected host to attack. It will then counter attack every host that it receives malicious packets from. The motivation behind this approach is to prevent any additional network traffic caused by an actively scanning anti-worm.

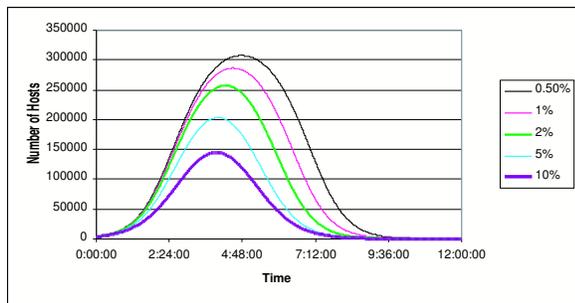


Figure 4: Passive Anti-worm Propagation with Different Initial Disinfection Percentages

The amount of cured hosts on the Internet will determine the success of this approach and therefore it is important to cure a set of hosts as soon as the anti-worm is available. This in turn requires an entrusted cyber-warfare organization which would have a list of hosts for the initial disinfection. The Internet Center for Disease Control (CDC) proposed by Staniford et. al. [28] can be used for the purpose. It is an Internet counterpart to the biological CDC.

Figure 4 shows the number of infected hosts for various initial cured host percentages between 0.5% and 10% (1,800-36,000 hosts). We observed that it takes a significant number of initially cured hosts to actually stop the spread of the malicious worm before it controls a majority of the network. The advantage of this approach is that it creates almost no additional network traffic, considering the fact that any message sent to a cured server would have been acknowledged if the service were running anyway. Therefore, the counter attack is not considered as an addition to the current traffic.

5.3 Active Scanning Anti-Worm

As described earlier, in a random scanning anti-worm scheme the anti-worm uses random IP scanning for target selection. We assumed that the anti-worm has the same number of threads and timeout value as the malicious worm, therefore, having the same scanning rate as the malicious worm. Our simulation, in Figure 5, shows that the anti-worm is successful in keeping the malicious worm below 15%, however, there are a couple of issues to be considered. The malicious worm effects the community in two ways: first by compromising and possibly damaging hosts, and secondly, saturating the network by creating excessive traffic. It is important for an anti-worm not to create the same network traffic that it is trying to prevent in the first place. Another shortcoming of such a scheme is that even when the threat is ended, the anti-worm would continue its scanning, immunizing more hosts which in turn add their own scanning traffic to the network. Figure 5 shows the number of cured, infected and susceptible hosts along with sum of the number of scans from both the malicious and the anti-worm.

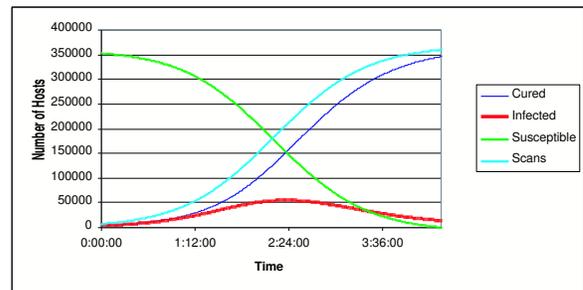


Figure 5: Active Scanning Anti-worm

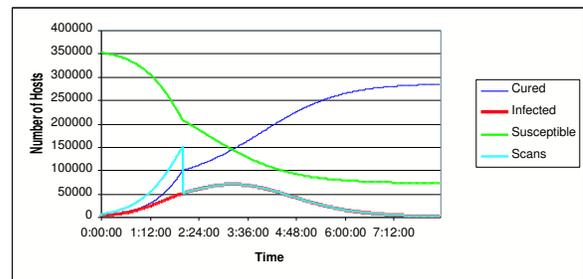


Figure 6: Active-Passive Hybrid Anti-worm (mode switch after 2 hours)

5.4 Active-Passive Hybrid Anti-Worm

We have also tried alternatives to a purely active scanning anti-worm. Since the malicious worm is suppressed after some time, the active scanning by the anti-worm may be stopped once the majority of malicious worm activities have stopped. This could be triggered by a simple timer, or more intelligently, by network traffic flow analysis. We have currently simulated a scenario in which the anti-worm stops active scanning after a predetermined time (currently two hours after its release) and then switches to passive mode. The traffic flow analysis method requires more future research. Even with the simple timer mechanism, we have observed the advantage of the hybrid method. Figure 6 shows the results. When the anti-worm stops active scanning (the vertical drop in the *scans* curve), anti-worm has propagated to twice as many hosts as the malicious worm (compare the *cured* and the *infected* curves at this point). The anti-worm is successful in keeping the malicious worm below 15% infected hosts, and the total scanning activity traffic is also kept low as can be observed from the *scans* curve.

5.5 IDS Based Anti-worm

The IDS based propagation scheme makes use of dedicated sensors deployed throughout the Internet to identify malicious attack vectors in the ongoing traffic. Once such packets are identified, both the sender and the intended recipient can be sent the anti-worm. We believe that scanning every ongoing packet is not feasible with current state of the art, therefore, we propose to use probabilistic inspection. We ran our simulations with varying capture probabilities and obtained some promising results. Figures 7 and Figure 8 show the effect of IDS-based anti-worm.

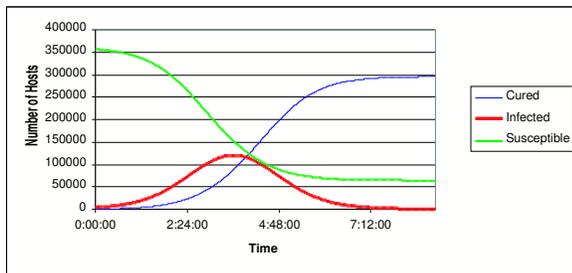


Figure 7: IDS-based Anti-worm Propagation (packet capture probability=0.05%)

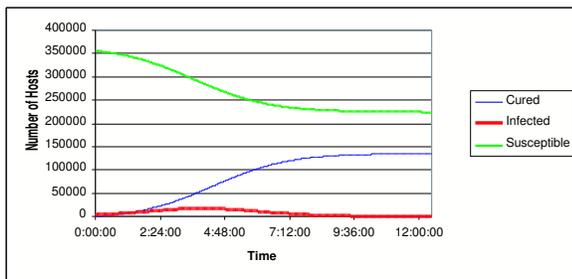


Figure 8: IDS-based Anti-worm Propagation (packet capture probability=0.2%)

Although the probabilities used lack a solid foundation, we believe that they are sufficiently low to be practically feasible. In fact, the probability of capturing a malicious packet depends on two factors: the fraction of packets that can be scanned by a sensor, and the expected number of sensors that a packet must go through to reach its destination. While the first factor is largely determined by current hardware technology, the second may be improved by the proposed Internet CDC with a better coverage of sensor deployment. The CDC will keep a list of IDS systems. With the combined effort of corporations and ISPs, it may be possible to form the basis for a powerful defense mechanism.

This method is inspired by the content filtering study carried out by Moore et. al. [19]. In that study, they simulate the effect of having content filtering in the top 100 ISPs. The results show that the infection can be kept fairly low with large response times. The complete inspection of all traffic at core routers, however, exceed the current capabilities of our technology. Our proposal instead assumes that only a fraction of the packets can be scanned, and simulation results show that the malicious worm can be effectively suppressed.

Through our simulations we show that it is possible to counteract a malicious worm even after significant time has passed since the initial infection. Therefore a CDC that incorporates IDS and our anti-worm counter-attack strategy has a significant chance of preventing more than 15% of the vulnerable hosts from being infected. Although a passive anti-worm is not quite effective on its own, it gives the anti-worm an advantage over the malicious worm - the ability to identify infected hosts. We find that the multi-state, initially actively scanning then passive, and the IDS based propagation schemes are very promising.

6. CONCLUSION

It is reasonable to predict that worms in the near future will be increasingly malicious and destructive, resulting in a new pervasive threat of information warfare. Because of this new threat, security agencies and the like must be prepared for the inevitable by employing appropriate countermeasures to combat the increasing threat. Recent outbreaks of the Code-Red, MSBlaster and SQL Slammer worms only reinforce the inadequacy of a system highly dependent on human factors to react accordingly. New defensive mechanisms must be invented and studied to better protect our systems. We have proposed the anti-worm generation framework in this paper. We did an preliminary study on automatically transforming a malicious worm into an anti-worm, on generating anti-worm code, and on the effect of the four different anti-worm propagation schemes through simulation. In spite of the limitations of the anti-worm in its current forms that will cause practical usage problems, we still find the idea an intriguing one. The techniques developed here (transforming a worm into an anti-worm with its own payload, and anti-worm propagation schemes in particular) would certainly be interesting to other researchers for studying future worms and for inventing new techniques.

7. ACKNOWLEDGMENTS

We would like to thank our shepherd, Nicholas Weaver, for his excellent comments and quick responses. We also thank the anonymous reviewers for their suggestions for improvement.

8. REFERENCES

- [1] C. Carver, J. Hill, J. Surdu, and U. Pooch. A methodology for using intelligent agents to provide automated intrusion response. In *Proceedings of the IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop, West Point, NY, June 6-7, 2000*, pages 110–116, 2000.
- [2] Z. Chen, L. Gao, and K. Kwiat. Modeling the spread of active worms. In *Proceedings, IEEE INFOCOMM*, 2003.
- [3] eEye Digital Security. Sapphire worm code disassembled. <http://www.eeye.com/html/Research/Flash/sapphire.txt>, January 2003.
- [4] M. W. Eichin and J. A. A. Rochlis. With microscope and tweezers: An analysis of the internet virus of november 1988. In *Proceedings of the 1989 IEEE Computer Society Symposium on Security and Privacy*, 1989.
- [5] J. Evers. Microsoft ponders automatic patching. *InfoWorld* (www.infoworld.com), August 2003.
- [6] R. Farrow. Reverse-engineering new exploits. *CMP Network Magazine*, 2004.
- [7] F. Gong. Next generation intrusion detection systems (IDS). *McAfee Network Security Technologies Group*, 2002.
- [8] H. HexXer. Codegreen beta release (idq-patcher/antiCodeRed/etc.). <http://www.securityfocus.com/archive/82/211428>, September 2001.
- [9] G. Hoglund. Buffer overflow construction kit. <http://www.rootkit.com/projects/winblock>, March 2000.
- [10] J. O. Kephart, D. M. Chess, and S. R. White. Computers and epidemiology. *IEEE Spectrum*, 1993.
- [11] J. O. Kephart and S. R. White. Directed-graph epidemiological models of computer viruses. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1991.
- [12] J. O. Kephart and S. R. White. Measuring and modeling computer virus prevalence. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1993.
- [13] M. Kern. Re: Codegreen beta release (idq-patcher/antiCodeRed/etc.). <http://www.securityfocus.com/archive/82/211462>, September 2001.
- [14] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. *2nd Workshop on Hot Topics in Networks (HotNets-II)*, 2003.
- [15] J. Leyden. Blaster variant offers 'fix' for pox-ridden pcs. http://www.theregister.com/2003/08/19/blaster_variant_offers_fix/, August 2003.
- [16] LSD. Dcom exploit. http://www.lsd-pl.net/files/get?WINDOWS/win32_dcom, 2003.
- [17] D. Moore. The spread of the code red worm. http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml.
- [18] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The spread of the sapphire/slammer worm. <http://www.cs.berkeley.edu/~nweaver/sapphire/>, 2003.
- [19] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet quarantine: Requirements for containing self-propagating code.
- [20] T. Mullen. Defending your right to defend: Considerations of an automated strike-back technology. <http://www.hammerofgod.com/strikeback.txt>, October 2002.
- [21] T. M. Mullen. Defending your right to defend: Considerations of an automated strike-back technology., October 2002.
- [22] R. Permech and M. Maiffret. Codered analysis. <http://www.digitaloffense.net/worms/CodeRed/code-red-original-eeeye/>, July 2001.
- [23] T. M. Project. Opcode db. http://www.metasploit.com/opcode_search.html, 2003.
- [24] N. Provos. A virtual honeypot framework. *CITI Technical Report 03-1*, October 2003.
- [25] J. Rapoza. Cheese: If it's good (worm), let it be! *ZDNet India*, July 2001.
- [26] G. Serazzi and S. Zanero. Computer virus propagation models. In *11th IEEE/ACM Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems MASCOTS*, 2003.
- [27] J. Shoch and J. Hupp. The worm programsearly experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, 1982.
- [28] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the internet in your spare time, 2002. To Appear in the Proceedings of the 11th USENIX Security Symposium (Security '02).
- [29] Symantec Corp. MSBlaseter, Symantec Security Response. <http://securityresponse.symantec.com/avcenter/venc/data/w32.witty.worm.html>, 2004.
- [30] Symantec Corp. MSBlaseter, Symantec Security Response. <http://securityresponse.symantec.com/avcenter/venc/data/w32.blaster.worm.html>, 2004.
- [31] Symantec Corp. Symantec Security Response. <http://securityresponse.symantec.com/>, 2004.
- [32] T. Vogt. Simulating and optimising worm propagation algorithms. <http://web.lemuria.org/security/WormPropagation.pdf>, February 2004.
- [33] A. Wagner, T. Dübendorfer, B. Plattner, and R. Hiestand. Experiences with worm propagation simulations. In *Proceedings of the 2003 ACM workshop on Rapid Malcode*, pages 34–41. ACM Press, 2003.

- [34] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. Large scale malicious code: A research agenda. *Silicon Defense Technical Report*, May 2003.
- [35] M. Williamson. Throttling viruses: Restricting propagation to defeat malicious mobile code. *HP Laboratories Technical Report*, 2002.
- [36] C. C. Zou, W. Gong, and D. Towsley. Code red worm propagation modeling and analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 138–147. ACM Press, 2002.
- [37] C. C. Zou, W. Gong, and D. Towsley. Worm propagation modeling and analysis under dynamic quarantine defense. In *Proceedings of the 2003 ACM workshop on Rapid Malcode*, pages 51–60. ACM Press, 2003.