DETECTION, PROPAGATION MODELING AND DESIGNING OF ADVANCED INTERNET WORMS

By

PARBATI KUMAR MANNA

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2008

To my family, friends, and teachers

ACKNOWLEDGMENTS

I want to take this opportunity to thank all the people who helped me during my doctoral sojourn. I understand that it is rather late to acknowledge their contributions, but as the saying goes, better late than never!

First, I want to thank my committee, starting with my advisor and Chair, Dr. Sanjay Ranka. He expressed his intention to work with me during my very first week of class at University of Florida, and has been a true guide to me in every aspect since then. He offered me complete freedom in pursuing my research in any area that I felt passionate about, and provided ample research direction from time to time. I am truly thankful and honored to work as his student for the past six years.

It has also been a pleasure to work with Dr. Shigang Chen, who served as my co-chair. A stalwart in the network research community, he has been instrumental in providing his domain expertise to my research area in a very big way. Without his help, I can barely imagine myself to be where I am now. I would also like to thank Dr. Alin Dobra, Dr. Christopher Germaine, Dr. Sartaj Sahni and Dr. Malay Ghosh who helped me in various academic as well as non-academic matters throughout my stay at Gainesville.

Finally, I want to thank my friends and family, without whose support I could have never lived through the ordeal of PhD. Special thanks go to my wife Madhuparna, who has been the mental driving power behind my efforts.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

DETECTION, PROPAGATION MODELING AND DESIGNING OF ADVANCED INTERNET WORMS

By

Parbati Kumar Manna

December 2008

Chair: Dr. Sanjay Ranka
Cochair: Dr. Shigang Chen
Major: Department of Computer and Information Science and Engineering

Malware, or malicious software such as viruses, worms, trojan horses or rootkits, pose a grave challenge to the computer user community by obtaining unauthorized access to computer resources. Among various malware, worms interest computer security researchers immensely due to their ability to infect millions of computers in a short period of time and cause hundreds of millions of dollars in damage. Unlike other malware, worms can replicate themselves over the Internet without requiring any human involvement, which makes their damage potential very high. Security researchers strive to prevent, detect and contain worms, as well as model their propagation patterns over the Internet.

Our work is primarily divided into three parts. The first part is geared towards devising a detection mechanism for an advanced worm called ASCII worm which has a very high damage potential due to its ability to compromise servers that are otherwise not vulnerable to common worms. The second part derives an exact analytical model for the propagation of permutation-scanning worms, a class of worms that employ a sophisticated propagation strategy called permutation scanning. The final piece of work re-examines the classical worm propagation models in light of the pseudo-random nature of the output generated by the random number generators used by the worms, and designs a worm that exploits the pseudo-randomness to achieve an optimal scanning strategy with high speed of infection, fault tolerance and low detectability.

Our work focuses on highlighting the damage potential of worms, and shows novel ways to detect them. It also provides accurate analytical propagation model for worms. This can help network security personnel to better understand the worms' spreading behavior, and design containment techniques and other countermeasures.

CHAPTER 1
INTRODUCTION

Security has been one of the primary concerns since the advent of computers. In fact, one of the first generation of the known computer viruses emerged in as early as 1970 over the ARPANET [36]. In that era, because not all the computers in the world were connected like today, the outbreaks of viruses were not pandemic. However, with the arrival of Internet, the computers all over the world were now networked and thus communicable. While this ubiquitous connectivity resulted in a great number of beneficial effects in the computer industry as well as in everyday life, it also had an unfortunate side-effect – now the problem of computer security had a new dimension called network security that needed to be addressed.

In a broad sense, network security comprises of the provisions and policies undertaken by network administrators to protect the underlying network and the network-accessible resources from unauthorized access, and the effectiveness (or lack) of these measures. Network security is subtly different from computer security, and we demonstrate the difference between the two using the following example from medieval history. Suppose our goal is to protect all the inhabitants (hosts) inside a fortress. Network security is akin to guarding all the entry points of the fortress, while computer security is comparable to providing armors to individual soldiers inside the fortress. It is evident that the former method is more powerful, because without it, we will have to rely upon the secureness of each individual hosts, which may not be a very good idea considering the heterogeneity of the hosts and their individual capability of defending themselves.

A network is under attack every day, and thwarting the attacks is the basic challenge of network security researchers. Examples of such attacks include obtaining unauthorized entry by fooling the authentication system, eavesdropping, and modification of the data to name a few. Most of the vulnerabilities that are exploited in such an attack are either cryptographic or resulting out of insecure implementation of policies. Malicious software

designed to infiltrate or damage a computer system without the owner's informed consent in such attacks are known as *malware*. It is a general term used to indicate a variety of forms of hostile, intrusive, or annoying software or code snippets. Examples of malware include computer viruses, worms, trojans horses, rootkits and many more. In this work, we focus on the worms.

1.1 The Computer Worm: A Brief History

Similar to its real-world counterpart, a computer worm (or Internet worm) replicates itself multiple times over the Internet, thereby rendering its target hosts compromised and the network congested. It has the ability to infect millions of computers in a very short period of time [42]. It is different from a computer virus in the sense that a virus requires human action to activate and popagate, while a worm is able to propagate by itself. However, it must also be noted that with a significant amount of malware being distributed via email nowadays, the distinction between the two is getting somewhat blurred. In fact, the very first case of replicated malware, the Christmas Tree EXEC Trojan horse [23] that brought down many IBM mainframe computers in 1987 spread using mass-mailing mechanism only, though it required human action for spreading. The true first case of a self-replicating worm causing significant damages is attributed to the Morris worm in 1988 [57]. Since then, both sophistication and damage potential of worms have increased tremendously, infecting millions of computer and causing hundreds of millions of dollars in damage. Notable mentions include Melissa (1999) [12], ILOVEYOU (2000) [10], Code Red (2001) [80], Nimda (2001) [11], Sapphire/Slammer (2003) [42], SoBig(2003) [14], MyDoom (2004) [15] and Zotob (2005) [54]. Symantec stated in their 2008 global internet security threat report [64], "Of the top 10 new malicious code families detected in the last six months of 2007, five were Trojans, two were worms, two were worms with a back door component, and one was a worm with a virus component."

1.2 Propagation Methods of a Worm

The way a worm propagates is as follows. Since by definition a worm code does not need to be executed manually by a human, it must compromise a program that is already executing. This is why in order to take over a host, worms attack the servers, programs that are always running and expecting input from other hosts (clients). A worm's behavior can be classified into two orthogonal categories: host-level behavior and network-level behavior. The host-level behavior comprises of its action on that target host, which may include infecting the host, modifying data, terminating or starting other programs, installing backdoors and Trojans etc. To analyze the host-level behavior, it is necessary to analyze the actual worm code, i.e. the payload. The network-level behavior is an Internet-scale picture of the worm traffic and is caused by a worm's act of self-replication, where it attempts to initiate connections to other potentially vulnerable hosts that could possibly be running the same server software that can be compromised. The network-level behavior is defined by the worm's propagation characteristics over the Internet, which tells how fast the worm spreads and how quick must any defense mechanism be in order to counter it. These two aspects of the worms behavior are discussed in further detail below.

1.2.1 Host-Level Behavior

First we take a closer look into how a worm compromises a host. Although there are exceptions, most of the worms employ a control hijacking technique known as *buffer overflow*, and the seminal paper by Aleph One in 1995 [3] describes in great detail how it can be achieved. A very brief overview of the process is as follows. Unlike Java, inherently unsafe languages like C and C++ do not consider arrays to be first-class objects and hence do not provide automatic bound checks for them at runtime. This allows a buffer to be overflown with a string longer than the buffer length, thereby overwriting the adjoining memory locations. Inside the runtime stack, the return address for the called procedure is located near the locations for the local variables of the called procedure (including the buffer), and is thus vulnerable to be overwritten by an overflowing buffer. If the return

address is overwritten with a new value, then when the procedure returns, it returns to the memory location pointed to by the new address. This process is aptly known as *control hijacking*. This new return address either directly points to the injected attack code in the buffer (method employing *NOP sled*), or points to some static address containing a instruction that causes the instruction stream to point back to the injected code. The latter method, mostly found in the static DLLs which are loaded at fixed locations in memory for Windows, is known as *register spring* [20]. In either case, the immediate effect is the execution of the attack code, with the final result of spawning a shell. Also, if the compromised application was running with root privilege, as many Windows server applications do, the attacker will now have access to a root shell.

1.2.2 Network-Level Behavior

The network level behavior of a worm, which is mostly influenced by its choice of propagation mechanism, is the Internet-scale picture of the network traffic generated by the worm. Generally, the goal of a worm is to infect as much of the vulnerable population as possible before malware detectors become aware of its presence and take countermeasures to block the propagation of the worm. Therefore, a worm must have a good *infection speed*, a term we use to indicate how fast a worm can infect the vulnerable host population. Paradoxically, high infection speed may also hinder the worm's propagation. To see why, we observe that predominantly a worm does not know beforehand which hosts in the Internet are vulnerable. Therefore, in order to infect the whole of the vulnerable population, it must send the attack code to arbitrary addresses over the Internet, a process known as *scanning*. It is evident that if the worm scans at a higher rate, i.e. sends the attack code to more number of target hosts within the same time period, then it will be able to infect the vulnerable host population faster. However, the overall network traffic generated by this increased scanning activity will also be higher, and this increased traffic may be noticed by a network administrator or malware detectors, thus revealing the presence of worm activity and causing countermeasures to be

taken. Worse, high amount of worm traffic may cause the network to be clogged or even partitioned, thereby hindering further propagation of the worm itself. Therefore, worms must maintain a fine balance between infection speed and detectability, and hence they must choose their spreading mechanism very judiciously.

One interesting observation is that the propagation efficiency of a worm depends not only on how fast it is scanning, but also how intelligently each actively scanning host is choosing its scan target addresses. Scanning strategy, i.e. the mechanism using which a worm chooses its target addresses to be scanned, has a significant impact on its network level behavior. It may select those addresses completely randomly (random scanning method), or it may choose to scan some portions of the Internet with a bias, e.g., once a vulnerable host is found, hosts belonging to the same subnet are scanned first (local subnet scanning). Or, the worm may also find a list of other communicable hosts from certain files of the infected host (like rhosts file in UNIX) and scan those addresses first (topological scanning). Although historically most worms choose the random scanning method or its variant, worms often deploy other methods of scanning as well, or a mixture of different scanning strategies. We would pay particular attention to one strategy called "permutation scanning" later.

While the network level behavior of a worm is dependent on the scanning strategy chosen by the worm author, it also depends on the properties of the network itself, i.e. the bandwidth, the link capacity, computational power of the routers etc., parameters that are not controlled by the worm author. A worm scanning at a very fast pace may put excessive burden on a router and thus cause it to go down or reboot, causing a partitioned network. While this does not help the worms propagation, it causes tremendous amount of damage by disconnecting multitude of hosts from the Internet, essentially performing a denial of service attack not only on e-commerce and personal activities but also on life-saving communication infrastructures. The slammer worm, which did not cause any damage to the individual hosts it infected, nonetheless caused hundreds of millions of

dollars in damage in cleanup activities [42]. Therefore, depending on the propagation strategy used, a worm can target either the hosts on the Internet, or the network itself. This makes the propagation strategy of a worm very important.

1.3 Recent Trends Among Worms

Although a significant amount of research has already been done in the various countermeasures against worms, as keeping with the paradigms in security research, a countermeasure is rarely foolproof forever and is only good until worm authors find a novel way to circumvent it. Thus, the landscape in worm research is constantly changing, and worm researchers will always have to play a cat-and-mouse game with worm authors, who are constantly upgrading their arsenal with innovative techniques and practices. Some of the alarming trends that have been observed in the recent past are discussed as follows.

1.3.1 Advent of Zero-Day Worms

The gap between the discovery of a vulnerability and the emergence of a worm exploiting that vulnerability is decreasing at an alarming rate. The Sapphire-Slammer worm (Jan 25, 2003) exploited a vulnerability which was discovered more than six months ago [42]. However, since then the gap has been decreasing, and starting from 2004 we have been seeing zero-day worms [60]. Considering the amount of damage Slammer was able to do in spite of its six-month lag, zero-day worms, which literally give no time for fixing the vulnerable systems, can cause very serious damage. This is also exacerbated by the fact that even when the vendor releases the software update (i.e. the "patch") that eliminates the vulnerability, a lot of computers do not get patched immediately due to various reasons, some of which are listed as follows. First, the user may be simply inactive, lazy or lacking the necessary bandwidth to download and install the patch. Second, many system administrators are wary to install a patch to the computers they manage before testing it adequately, as the patch might break existing applications, and rollback is often difficult, time consuming, or simply not possible. Third, the patch might only be available for the genuine copies of the software, as was the case for Windows XP service packs,

which was a number of patches bundled together into a single software upgrade. These service packs were available only through the auto-update option for the genuine copies of Windows XP, which means many of the vulnerable computers, which used a pirated copy of Windows XP, would not get these updates.

1.3.2 Emergence of Polymorphic Worms

Since their inception, worms have been showing an increasing trend of resisting detection by using various evasion and obfuscation techniques. In fact, the very first Internet worm [57] deployed encryption techniques in order to hide its code. Of late, we have been seeing more and more instances of polymorphic worms, where the worm changes its code every time it infects a host. Also worms with self-mutating code, where the original code modifies itself during execution to generate a completely new code body, are becoming more commonplace [24, 1]. These kind of worms are very hard to detect, since different instances of the same worm share few similarities. Commercial malware detectors, which in most of the cases rely upon substring matching to detect a malware, are often ineffective against such kind of attacks. In this dissertation, we will develop detection strategies for one such worm, viz. the ASCII worm.

1.3.3 Arrival of *Script Kiddies*

Another worrying trend is that the task of creating a new worm, which previously was limited to only a group of selected individuals with the so-called "hacking" skills, has now been made considerably easier with the abundant supply of worm-creating frameworks over the Internet [24, 1]. With these frameworks, it is now possible for a common person with very limited knowledge (called the *script kiddy*) to create and release a worm in the wild. Also, the worm authors now use modularization techniques in their programs so that attack components can be readily added or substituted. This has exacerbated the problem of creating new worms and their variants.

1.3.4 Shift in Hackers' Mindset

The last trend is the most worrying one – it is regarding the changing mindset of the worm-author from being a prankster to a fraudster. Originally, worm authors were interested to create worms that would do visible damage, and sheer joy of creating global nuisance and making a name for themselves seemed to be the intended goal. However, due to many reasons including successful prosecution and incarceration of worm-authors, of late the focus has shifted from prank to money. Nowadays hackers are more interested in creating worms that compromise a host without any perceivable changes to the user, so that the host can serve as a zombie of a botnet, which is the network of compromised computers under the control of the attacker. Worse, since there are no uniform laws regarding cyber crimes across the globe, it is often hard if not impossible to prosecute an attacker residing in a distant foreign country. Thus, while worm attacks nowadays seldom create as much newspaper headlines as they used to do at the beginning of the current decade, reports published by anti-malware industries confirm significant and sustained worm activity [64].

1.4 Key Challenges in the Worm Research Area

As a result of the recent developments stated above, we are starting to observe more and more zero-day worms armed with different evasion methods including encryption, polymorphism and mutation. Moreover, due to significant amounts of research being done regarding efficient propagation strategies, todays worms enjoy having access to an abundance of smart propagation mechanisms that can be easily harnessed. Deploying such advanced propagation strategies enables a worm to significantly reduce its network footprint, i.e. the total amount of scan traffic generated over the Internet. If the worm keeps it scanning rate low, then such a worm would be very difficult to detect by monitoring traffic surge on the Internet alone. Thus, in order to thwart this new threat of advanced worms, worm researchers need to keep up with the latest developments, which

opens up ample research opportunities. Some of the basic challenges in worm research, details of which are presented in chapter 2, are:

- **Worm Prevention:** How do we prevent worm outbreaks?
- **Worm Detection:** Considering the availability of advanced obfuscation strategies like encryption, polymorphism and mutation, how do we detect worms?
- **Worm Containment:** Assuming that prevention and detection strategies do not always work, how do we contain an ongoing infection and minimize the damage?
- **Worm Propagation Modeling:** For the different propagation strategies deployed by worms, how do we assess their effects on the Internet?
- **Worm Design:** How do we design a worm that has an optimal scanning strategy, i.e. it achieves the right combination of high infection speed, resilience to cure and low network footprint?

Since different worms deploy different infection strategies, it is difficult to envision a universal countermeasure that will be equally applicable, and effective, against all kinds of worms. Similarly, because of the heterogeneity in the espoused scanning strategies, a single propagation model cannot describe the traffic pattern generated by all possible worms. Therefore, the detection methods and propagation modelings must be done for each class of worm on an individual basis. In this dissertation, we address the following specific problems in the area of detection, propagation modeling, and design, respectively:

1.4.1 Worm Detection

As mentioned earlier, in order to evade malware detectors, today's worms employ encryption, mutation and various other obfuscation techniques. ASCII worm, whose body consists of entirely text. i.e. printable ASCII characters, is an example of such a self mutating worm. It is very appealing since it can sneak in places where usual worms cannot. For example, since a large number of network protocols (or parts thereof) are text-based, at times the servers based on those protocols use ASCII filters to allow text input only. However, simply applying ASCII filters to weed out the binary data is not enough from the security viewpoint since the assumption that malware are always binary

is false. As we will demonstrate, although text is a subset of binary, the effectiveness of binary malware detectors is severely dwindled for detecting text malware. We investigate the threat posed by an ASCII worm and design an efficient scheme for its detection.

1.4.2 Worm Propagation Modeling

The scanning strategy chosen by a worm determines its infection speed, i.e. how fast it can infect the whole of the vulnerable host population, and its stealthiness, i.e. how much less network footprint it generates. An accurate analytical propagation model is an important tool in understanding the threat posed by a worm, since it also allows us to comprehensively study how a worm propagates under various conditions, which is often computationally too intensive for simulations. More importantly, it gives us an insight into the impact of each worm/network parameter on the propagation of the worm. Traditionally, most modeling work in this area concentrates on the relatively simple random-scanning worms. However, modeling the propagation of permutation-scanning worms [63], a class of worms that are fast yet stealthy, has been a challenge to date. We attempt to solve this problem here.

1.4.3 Worm Design

We have already observed that worm authors need to maintain a delicate balance between infection speed and stealth in order to create maximum impact with minimum alerts generated. Therefore, an optimal scanning strategy would be one that allows a high infection speed while keeping a low network footprint. While searching for the optimal scanning strategy, we stumble upon a very important discovery: the output of the random number generator used by a worm is pseudo-random and not truly random, and this observation raises doubts about the validity of the derivation of the epidemic model used for explaining random-scanning worms. We attempt to derive the correct model, and investigate if this pseudo-random property can be exploited to make the worms scanning optimality.

We note that only the rudimentary details of the proposed work is given here. Once we discuss the state of the current research in Chapter 2, the undertaken work will be presented in greater detail in Section 2.5.

CHAPTER 2
RELATED WORK

Since a worm has a very high damage potential, the main challenges facing worm researchers are how to detect, stall and prevent their spread. Correct modeling of their propagation is also very important since it gives an idea of how fast the worm can spread and how quickly a defense must react to mitigate the damage by stalling its advancements. Thus, correct propagation modeling may not be a direct line of defense in thwarting a worm, but it is a very important tool in aiding other countermeasures and helping to make policy decisions. The worm research is traditionally focused mostly on the following countermeasures: prevention, detection and containment, along with propagation modeling. A survey of the related work in each of the research areas is presented below.

2.1 Prevention

The goal of prevention is to make sure that the hosts are not vulnerable to worm attacks in the first place, and the focus is on ensuring that either the code or the infrastructure executing the code is secure. The first approach includes espousal of good programming practices of writing secure code. The other approach assumes that the server code is inherently vulnerable, and hence strives to fortify the code execution infrastructure of the server in such a way that even if a vulnerability is targeted by a worm, the server is still not compromised. For example, any change that makes the buffer overflow impossible would automatically reduce the number of worm attacks significantly. Some of the preventive methods that use infrastructural modifications to thwart a worm using buffer overflow techniques are discussed below.

StackGuard [19] proposed using a canary beside the return address on the stack to detect if the return address has been overwritten by checking if that canary has been changed. However, it still failed to protect other function pointers and heap corruption. Pointguard [18] went one step further by keeping track of each pointer. However, it could still be broken using brute force approach in some cases. The Address Space

Randomization technique [4] suggested randomization of the address of the stack and various other libraries, thus making the job of guessing the return address much harder for the attacker. There have been other hardware-based approaches like Return Address Stack [49, 25] that attempted to detect if the return address has been overwritten. Also, as most of the overflows happen on the stack (heap is also another alternative), making the stack non-executable also appears to be an immediate solution, since very few applications actually require the stack to be executable by design. This approach was proposed by Solar Designerand implemented as the NX patch in the Linux kernels in Red Hat Fedora Core 2 as of build 2.6.6-1.427 [5]. Similar prevention method has been offered by Windows XP SP2 as an optional feature as DEP (Data Execution Protection) in both hardware and software level (if the added NX bit in the page table is not yet available by the processor). The same feature is called EVP (Enhanced Virus Protection) in AMD64 architecture [79]. We note that most of these prevention techniques are host-based and hence dependent on the proper implementation on the user's part for it to function properly. Therefore, these kind of prevention techniques can be thought of as the last line of defense against the computer worms, where the worm is thwarted just when it is about to compromise its victim.

2.2 Detection

A significant amount of worm research focuses on timely detection of worm, with good precision and recall. The worm may be detected at many stages: when the scan packets are being sent (extrusion detection), at the border router of the destination network, at the network layer of the target host, at the application level of the target host, or even during execution of the malicious code. However, the distinction between prevention and detection becomes blurred for some cases. For example, once StackGuard detects a buffer overflow, further execution of malicious code is prevented, which makes it is unclear whether its action should be categorized under prevention or detection. Nevertheless, when the malicious traffic is detected at the network level itself, it is much

easier to classify the countermeasure as pure detection. For detection at the network-level, the Intrusion Detection System (IDS) may take into account not only the payload of the worm but also the network traffic data such as volume surge, nonexistent destination hosts etc. A survey of selected network-level detection methods is given below.

We start with vulnerability-specific detection methods. It is a fact that for a worm to spread, an exploitable vulnerability must exist. Usually, this vulnerability lies on a *specific* branch of control flow which is less traversed, as otherwise the chances of the vulnerability getting detected during the software testing phase is higher. Therefore, to ensure that the control traverses through that specific route, the worm must have vulnerability-specific input data. For example, for the LSASS exploit, one must have the "\PIPE\lsarpc" string and a particular field of a logged record long enough for the buffer overflow to occur [20]. Shield [71], which models vulnerability signatures consisting of all possible sequences of application messages and payload characteristics that would lead to any remote exploit of that vulnerability, is based on this idea. Another detection method is Vigilante [17], which upon information of an attack generates vulnerability-specific self-certifying alerts that can be distributed among hosts to warn about the danger and expecting them to use vulnerability-specific filter.

Examining the content and underlying structural properties of the malicious payload is the key in detection of worms for many IDSs. For example, many detection mechanisms including Earlybird [61], Autograph [30] and Polygraph [44] are based on the premise that different instances of a zero-day worm would contain common substrings or fingerprints, which would potentially have vulnerability-specific patterns. Buttercup [50] tries to find the range of Return Addresses, which usually forms part of the string which is used to overflow the buffer. Unfortunately, since it must know the range of the Return Addresses apriori, it only works for known worms. Abstract Payload Execution [67] hypothesizes that the presence of a NOP sled would lead to a long sequence of valid instruction stream, which is uncharacteristic of a random data. However, since NOP sleds are hardly used any

more, this method will miss today's worms that do not use the NOP sled and instead uses the Register Spring technique [20]. STRIDE [2] also employs similar technique to find the existence of NOP sleds in polymorphic worms. There are also other detection strategies that involve inspecting the structure of the payload and finding anything anomalous. In Styx [9], Chinchani et al make the following observation that normal (benign) data is different from a program code fragment, which has got a definite control and data flow. Styx tries to detect the worms by identifying those program structures, by creating CFGs (Control Flow Graphs), which were first introduced by Kruegel et al [34] for designing the fingerprint of polymorphic worms. SigFree [73], a zero-day worm blocker also does not rely on any signature but depends on the presence of executable code inside network data flow. It claims to have the capability to detect ASCII worms; however, it usually bypasses the ASCII traffic since processing it would degrade the performance significantly. Emulation method [51] contends that with advanced polymorphic engines, different instances of the same worm will hardly have any common strings, and network-level emulation is the only way to catch a worm. PADS [65] uses expectation maximization techniques to detect the presence of decrypter in a polymorphic worm by generating a statistical signature of the worm.

There have been other detection techniques that focus more on the anomalous behavior of an infected host to detect worm activity. For example, since a random-scanning worm has little knowledge whether its target host actually exists or not, Honeycomb (2003) [33] exploits this by deploying *honeypots*, i.e. decoy computers, to generate signatures by detecting patterns in the traffic seen on the honeypots, with the assumption that *any* traffic destined towards non-existent hosts is malicious. The signature generation involves both behavior (protocol) analysis and substring matching of the payload, often requiring human intervention. One shortcoming of this method is that non-malicious traffic may accidentally hit the honeypots, creating noise in signature generation. Other drawback of this method is that creation of a good signature often requires a large number

of samples, which may take a long time to acquire considering the fact that the same honeypot may be targeted by different malware, and by the time the traffic is filtered for a single worm and the signature is prepared, the damage could be already be done. DOME [55] assumes that any dynamically generated or obfuscated code is malicious, and proceeds to statically preprocess the Windows executable to identify the location of the Win32 API calls relative to the executable, and raises alarm if the API call is made from a different location at runtime.

2.3 Containment

This approach assumes that it is not possible to prevent infection by worm completely, and hence a more practical strategy would be to focus more on *damage control*, i.e. slow down the spread of the worm so that not only the overall damage is mitigated, but also more time is available for researchers to come up with a countermeasure. Most common initiatives include throttling the scanning rate, limiting the number of outbound connections etc.

Moore et al. investigated the effectiveness of worm containment technologies (address blacklisting and content filtering) and concluded that not only such systems must react very fast, but also nearly all ISPs must employ the content filtering for it to be successful [43]. Park et al studied worm containment methods in power-law Internet topologies and with partial deployment [47]. Williamson et al proposed to modify the network stack to bound the rate of connection requests to distinct destinations [77, 68]. This method has the disadvantage that it restricts a normal host the same way as a worm-infected host. Also, in order to succeed, this approach must be adopted universally, which is a very unrealistic requirement. Weaver et al showed that fast containment of random scanning worms on a large scale network [76] is possible using the Threshold Random Walk (TRW) algorithm first introduced by Balakrishnan et al in [29]. Schechter et al [58] proposed a credit-based algorithm to limit the scan rate of a host, whose credit (i.e., allowance of making connections) is increased by one for each successful connection

made and decreased by one for each failed connection made. This algorithm can once again be circumvented by an infected host that scans while making legitimate connections at the same rate. DAW [6] inspects the failed connection statistics for the hosts inside an ISP and employs spatial and temporal rate-limit algorithm (limiting the number of failed connection attempts) to slow down the worm based on the DAW parameters, not the worm parameters.

2.4 Propagation Modeling

In the real world, the knowledge of an outbreak is a necessary criterion for the subsequent development of a cure for the unknown disease. In the perspective of network security, one must first be aware that there is an worm outbreak in order to analyze the malicious traffic and then prepare a suitable patch. The worst kind of worm is one that is never detected, since such a worm can do extensive damage without the user ever being aware of it. However, even though such a worm will leave little evidence about their presence on the target hosts, in order to propagate, it will still generate a surge in the network traffic in the destination port. Therefore, one way to detect such an outbreak could be through detection of a surge in the global network traffic on specific port(s). Since noise is a significant factor in the Internet traffic, one must know the exact propagation characteristics of a worm to decide whether there is any worm activity or not. Due to these reasons, propagation modeling remains a very important tool in the war against worms.

We describe some of the important propagation models as follows.

- ***Classical Epidemic Model (the SI Model)***
  In classical simple epidemic model, there are only two possible states for each host: susceptible (vulnerable) or infectious. Since infection attempts have no effect on the non-vulnerable hosts, they are not considered in this model. The model also assumes that once a host is infected, it stays in that infectious state forever – there is no "infected but not infectious" state. Thus state transition of any host can only be: `susceptible` → `infectious` [28]. The classical simple epidemic model for a finite

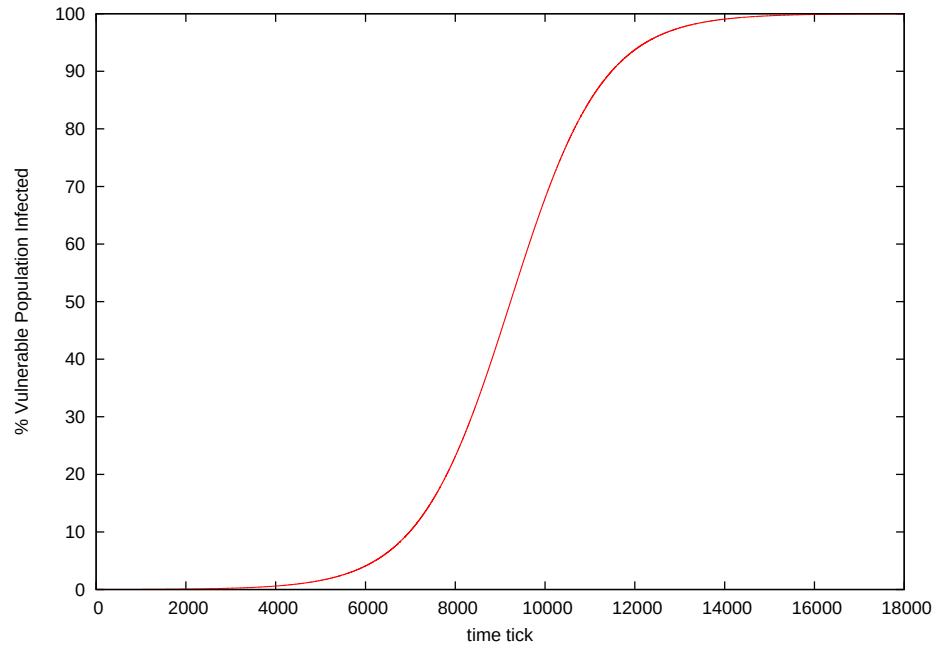

Figure 2-1. Classic epidemic model of propagation of contagion

population is

$$\frac{dI(t)}{dt} = \beta I(t)\ [V - I(t)] \tag{2–1}$$

Dividing both sides by V, we obtain

$$\frac{d\bar{I}(t)}{dt} = \beta V\ \bar{I}(t)\ [1 - \bar{I}(t)] \tag{2–2}$$

where $I(t)$ and $\bar{I}(t)$ are the number and the fraction of infected hosts at time t respectively; V is the size of vulnerable host population; and β is the pairwise infection rate. At beginning, $t = 0$, $I(0)$ hosts are infectious and the other $V - I(0)$ hosts are all susceptible. Solving the differential equations for $V = 2^{13}$, $I(0) = 1$, and $\beta = \frac{1}{2^{10}}$, we obtain a sigmoid curve as in the Figure 2-1. The beginning of most of the worm epidemics matches the early part of this curve, until the effects of bandwidth limitation and network congestion set in. This was demonstrated by Staniford et al [63] for Code Red worm.

- ***Generalized Epidemic Model (the SIR Model)***
  Kermack-Mckendrick extended the simple epidemic model by considering the removal process of infectious hosts [28]. In this extended model, it assumes that during an epidemic, some infectious hosts either recover or die; however, once a host recovers from the disease, it attains perpetual immunity to the disease. The hosts that recover or die from the disease are put in the "removed" state (which is an

addition to the simple model). Thus each host can be in only one of following three states at any time: susceptible, infectious, or removed. Any host in the system either undergoes the state transition "`Susceptible` (S) → `Infectious` (I) → `Removed` (R)" or stays in "susceptible" state forever. In this model, $A(t)$ and $R(t)$ denote the number of hosts that are infectious and removed at time t. Now, let $S(t)$, $I(t)$ and $R(t)$ denote the number of susceptible, infectious and removed hosts by time t. Evidently, $V = S(t) + I(t) + R(t)$. We obtain the following propagation equations:

$$\frac{dS(t)}{dt} = -\beta I(t)S(t) \quad (2\text{–}3)$$

$$\frac{dI(t)}{dt} = \beta I(t)S(t) - \gamma I(t) \quad (2\text{–}4)$$

$$\frac{dR(t)}{dt} = \gamma I(t) \quad (2\text{–}5)$$

where V is the vulnerable host population size, β is the pairwise infection rate and γ is the rate of removal of infectious hosts.
One can see that from the model that in order for an epidemic to happen, the number of infectious hosts must rise initially. Thus, at the beginning, we must have $\frac{dI}{dt} > 0$, which implies $\beta I(t)S(t) > \gamma I(t)$, or $S(t) > \frac{\gamma}{\beta} I(t)$. This $\frac{\gamma I(t)}{\beta}$ is known as the epidemiological threshold, which is defined as the number of secondary infections caused by a single primary infection. Stated differently, it determines the number of people infected by contact with a single infected person before his death or recovery. When this threshold is below 1, each person who gets infected will infect fewer than one person before recovering or dying, so the outbreak will eventually wane.

- ***Realistic Scenario – Two-Factor Worm Model***
  Although the Kermack-Mckendrick model does extend the original SI model, there are a few reasons it cannot be applied to Internet straightaway. First, the Kermack-Mckendrick model considers removal of infectious hosts only, while in reality patching, filtering and similar countermeasures remove both infectious and susceptible hosts from the total vulnerable population. Moreover, this model assumes that the infection rate is constant, which is not necessarily true for a rampantly spreading worm like Code Red. The reason for the latter event is that large-scale worm propagation causes excessive load on the routers, sometimes even overwhelming them as their ARP caches fill up. And once one router goes down, other routers need to recompute new paths and update their routing tables, which again causes even more load, and further degradation of performance. While the degradation of performance of routers and the resulting disruption of service may very well be one of intended result of the worm, this also has a negative effect on the worm propagation itself. Because now not all scan packets would reach their destinations, the overall scanning rate (and thus scanning efficiency) of the worm decreases.
  We do not pursue the details of this model any further as the goal of this model is to obtain the propagation curves in presence of *practical* constraints like congestion and

recovery. However, these conditions do not necessarily apply, or hinder, the progress of a smart worm that scans at a rather slower rate and is thus able to reduce its network footprint drastically.

2.5 Contributions

In this section, we give a brief description of the research work that has been undertaken in this dissertation. The research problems described here will be discussed in more detail in the following three chapters.

The broad objective of this research is to analyze the host-level and network-level behavior of today's worms, which are equipped with latest evasion and obfuscation tools and intelligent scanning strategies, and devise possible countermeasures. Since different worms use different strategies, evidently it will be implausible to devise a generalized defense strategy that will be effective for all possible kinds of worms. Therefore, instead of proposing a cure-all solution, we attempt to solve the following specific problems that highlight some of the prevalent threats posed by worms today:

2.5.1 Detection of ASCII Worm

ASCII worms, i.e. worms whose body consists of entirely text, or printable ASCII characters, are one of the latest generations of the self-mutating worms. It was shown in as early as 2001 that it is trivial to convert a binary worm into an ASCII worm [56](by binary, we mean containing both printable and non-printable characters). They are very appealing since such a worm can obtain access to places where a worm is not expected to be able to get in under normal circumstances. For example, there are cases where a server expects certain kind of traffic to be strictly text, which is in fact quite common as many important applications work with protocols that, or parts thereof, are text-based. Examples of such text-only traffic include the URL in a HTTP request, or the email traffic. To ensure that only the text characters get in at times when text is expected, these servers usually employ ASCII filters [26] which drop any binary input. This filtering results in a beneficial side effect of eliminating worms that exploit any possible vulnerability in the execution paths for processing the input, since worms are

usually binary. However, using the ASCII filter alone as the sole defense against malware is dangerous, since the malware may very well be text-based. Thus, we believe that the text stream should not be bypassed from any scrutiny reserved for binary, and should be treated with equal suspicion. There are other cases too where the ASCII stream is not bypassed, but the detectors find the ASCII stream yielding too many false positives due to their structural properties. Therefore, efficient detection of ASCII worms remain a challenge till date.

2.5.2 Exact Modeling of the Propagation of Permutation-Scanning Worm

Unlike a real-world contagion which spreads in all directions randomly, a computer worm has control over its choice of the next target host to be scanned. Most of the worms till date use random scanning or its variant, where each infected host chooses its next potential victim randomly. Among the various intelligent scanning strategies available, Permutation-Scanning [63] features as one of the most interesting strategies. In that strategy, the infected host uses permutation to map the real address space into a virtual one, which effectively causes the vulnerable hosts to be dispersed evenly in the virtual address space, even if they were present in clusters in the original address space. Initially a small number of infected hosts start scanning the addresses sequentially after their own addresses on this virtual address space. Whenever any of them infects a new vulnerable host, it continues to scan sequentially, while the freshly infected host chooses a random location on the permuted address space and starts scanning sequentially from there in the same direction. After hitting an already infected host, the scanning host may either choose to retire or select yet another random location on the permuted space to resume its scanning. Simulations show that while this propagation strategy has a high infection speed, it also causes significantly less network traffic compared to a randomly-scanning worm, and hence is much stealthier. Therefore, modeling the propagation of this worm is very important, since without the knowledge of its exact propagation characteristics, it has the potential of passing undetected.

2.5.3 Worm Design: Exploiting Pseudo-Randomness for Optimizing Scanning Strategy

The virulence of a worm is indicated by how quickly, resiliently and stealthily it can comprise the entire vulnerable host population, and it is dependent on the scanning strategy chosen by the worm. Different scanning strategies yield different results in terms of 1) Infection speed, which indicates how quickly the entire vulnerable host population is infected, 2) Stealth, which indicates how much network scanning traffic is generated, and 3) Fault tolerance, which indicates the ability to infect the vulnerable population completely in spite of failure of certain scanning hosts. Most worms till date use random scanning or its variant; however, some employ differential scanning, where different destination address blocks are scanned with different probabilities. An example of differential scanning is local subnet scanning, where based on the assumption that more vulnerable hosts will be clustered in the vicinity of the currently infected host, the addresses in the same subnet of the infected host are scanned with a higher probability. Our goal is to find the optimal scanning strategy that achieves all these goals, i.e., it ensures high infection speed with low network footprint and high fault tolerance. Since most random-scanning worms use a pseudo-random number generator (PRNG) to produce the sequence of target addresses to be scanned, we first investigate whether the classical epidemic model, which is universally accepted as the propagation model for such worms, takes into account the pseudo-randomness of the output from the PRNG. Surprisingly, we find that it does not, and this discovery casts reasonable doubt about that correctness of the derivation process of the classical epidemic model. We attempt to provide the correct derivation, and explore whether the pseudo-randomness property can be exploited to make the existing random-scanning worms much stealthier without losing infection speed and fault tolerance, thus progressing towards the elusive optimal scanning strategy.

CHAPTER 3
DETECTION OF THE TEXT MALWARE

In the past decade, the Internet has witnessed the rapid evolution of various malware (virus, worm, Trojan, to name a few) [64]. While a considerable amount of research has been devoted to the detection of the classical binary malware, the possibility of using purely text stream (keyboard-enterable, Hex 0x20 through 0x7E) as the carrier of malware has remained under-researched and often underestimated. Rix [56] and Eller [26] showed a few years ago that any binary code can be turned into functionally equivalent text (or even alphanumeric) code. Having a malware that is completely text-based is very appealing to the malware authors since it can open new attack channels that were earlier assumed to be malware-resistant simply by virtue of accepting text-only input. Today, many popular protocols or their components are text-based, e.g., HTTP requests, HTML, XML, or email traffic. To ensure text-only input, these servers often employ an ASCII filter to discard or mangle the binary input [26]. However, if the filter is the only defense, then these servers remain vulnerable, as the assumption that all malware are binary is false. Worse yet, even some malware detectors deliberately bypass text streams. For example, SigFree usually does not process the text-only input to avoid performance degradation [73]. Thus, the notion of regarding the text data as benign and not subjecting it to malware detection is dangerous, and we believe text should undergo the same scrutiny as binary.

Even when the text input is examined, today's malware detectors are not adequately suited for efficiently detecting text-based malware due to the structural properties of text. We consider two popular detection schemes: 1) disassembling the input into instructions and then checking for the validity and executability of the instruction sequence (e.g. APE [67]), and 2) examining the frequency distribution and other statistical properties of the payload (e.g. PAYL [72]). The first scheme has two problems. First, almost *any* text string translates into a syntactically correct sequence of instructions, which means

checking for syntactic validity is of little value for detecting text malware. Second, since most branch (jump) opcodes are text, the proportion of branch instructions in text data is significantly higher than that in binary data. Since each branch instruction forks the current execution path into two directions, having a lot of them exponentially increases the total number of paths to be inspected by a detector that checks for executability. In other words, to ensure quick detection of malware in text data, one must find novel ways to prune the number of execution paths to be inspected. Regarding the other scheme that examines the frequency distribution and other statistical properties of the payload, there are instances where text malware has been shown to successfully evade such detectors. For example, Kolesnikov *et al* [32] showed a way to create a text malware that follows normal traffic pattern to the extent that it can evade even a robust payload-based detector like PAYL [32]. Finally, we have performed experiments using a commercial malware detector to scan various binary malware and their text counterparts. Although the detector successfully catches all binary malware, no alarm was raised for the text. Therefore, we conclude that the threat of text malware is real, and we can ignore them only at our own peril.

ASCII worm, a worm whose body consists of entirely text data, is an example of a text malware. While we focus on detection of ASCII worms, we note that the detection techniques developed by us will be equally applicable for any text malware, not just ASCII worms. Therefore, throughout this chapter, we will mostly be using the term "text malware" rather than just ASCII worm.

In this chapter, we analyze the potentials and limitations of text malware, and formulate detection techniques that exploit those inherent limitations. We introduce a novel text-malware detection method that examines the maximum executable length (MEL) of the byte stream arriving at a server which runs protocols expecting text input. MEL measures the number of instructions on the longest error-free execution path in the disassembled input. Because of the inherent randomness in the disassembled instructions,

a benign stream of text is very likely to cause an error during runtime and thus not likely to have a long error-free execution path.

The concept of MEL was originally introduced in Abstract Payload Execution (APE) [67] for detecting binary worms. It raises an alarm when the MEL measured from the input stream is greater than a threshold value. However, we will demonstrate through analysis and experiments that APE, as well as other binary detectors, are not suitable for detecting text malware. Not only is it extremely slow due to an excessive amount of branch instructions in text input, but also its MEL measurement is incorrect without taking the text-specific properties into consideration. We further show that, even for binary worms, APE may no longer be effective because malicious binary code can be made very compact with such a small MEL that will overlap with the MEL range of benign input. On the other hand, as one of our contributions, we observe that it is very hard to make text malware compact due to the unavailability of critical instructions in the text domain.

We make two major contributions in this chapter. First, as the existing MEL-based detectors, in their current form, are unsuitable for text malware, we must explore new text-specific properties that characterize more precisely the structural limitations of the instructions in the text domain, which will in turn constrain how the text malware can be constructed. By exploiting the limitations and constraints of the text malware, we design DAWN (Detecting ASCII Worms in Networks), a novel MEL-based method for detecting not only ASCII worms but any text malware. It is fast, reliable and accurate. Second, how big should the MEL threshold be for raising alarms? In the past, the MEL threshold used to separate benign data and worms is obtained empirically. It does not explain whether there is a mathematical foundation behind the method, i.e. whether it is possible for a benign instruction stream to have an MEL higher than a given threshold, and if so, with what probability. We develop a probabilistic model of the MEL theory. We show how the MEL threshold can be calculated from the input character frequencies (instead of from

some training data [67, 2] that may be biased), and how we can control the detection sensitivity.

The rest of the chapter is organized as follows. Section 3.1 gives an overview of the text malware. Section 3.2 looks into the limitations of text malware and devises an MEL-based detection method exploiting those limitations. Section 3.3 derives the underlying probabilistic model for the MEL method. Section 3.4 describes the design and implementation of our text malware detector. Section 3.5 evaluates our detector through experiments. Section 3.6 provides a comparison of our work with others. Section 3.7 explores text malware in non-Intel architectures. Section 3.8 concludes with discussions on the limitations and robustness of our detector.

3.1 Inside the Text Malware

In this section, we begin with definitions and terminologies, then discuss the opcode availability in the text domain, and finally present the typical construction of a text malware. Our discussions are made in the context of the Intel 32-bit architecture. We will explore the possibility of text malware in non-Intel architectures in Section 3.7.

3.1.1 Definitions and Terminologies

We use the following definitions throughout this paper.

- **Text data**: byte stream consisting of only keyboard-enterable characters, or in other words, printable ASCII characters (0x20 through 0x7E). A malware whose instructions consist of only bytes in the ASCII domain is called a *text malware*. Throughout this paper, the terms "text" and "ASCII" will be used interchangeably.
- **Binary data**: byte stream consisting of text as well as non-text characters. A malware whose instructions contain bytes outside of the ASCII domain is called a *binary malware*.
- **Valid (or invalid) instruction**: an instruction that will not (or will) cause the running process to abort by raising an error during its execution.
- **MEL (Maximum Executable Length)**: length of the longest sequence of consecutively-executable valid instructions in an instruction stream. To elaborate, suppose we are given n bytes. We begin the disassembly from the i-th byte and use a count c_i to record the number of instructions that we can execute before an error

(i.e., invalid instruction) occurs. We repeat the procedure for $i = 1, 2, ..., n$, and the MEL is be given as $\max\{c_i, 1 \leq i \leq n\}$.

3.1.2 Opcode Availability for Text Malware in Intel Architecture

Most malware needs to make system calls, for example, when they open sockets to communicate with their masters or trying to propagate in case of worms. However, the opcodes required for those system-level functions are not available in text. The only Intel opcodes and instruction prefixes available in the text domain are:

- Dual-operand register/memory manipulation or comparison opcodes: *sub*, *xor*, *and*, *inc*, *imul* and *cmp*
- Single-operand register manipulation opcodes: *inc*, *dec*
- Stack-manipulation opcodes: *push*, *pop*, and *popa*
- Jump opcodes: *jo*, *jno*, *jb*, *jae*, *je*, *jne*, *jbe*, *ja*, *js*, *jns*, *jp*, *jnp*, *jnge*, *jnl* and *jng*
- I/O operation opcodes: *insb*, *insd*, *outsb* and *outsd*
- Miscellaneous opcodes: *aaa*, *daa*, *das*, *bound* and *arpl*
- Operand and Segment override prefixes: *cs*, *ds*, *es*, *fs*, *gs*, *ss*, *a16* and *o16*

While many critical instructions for system calls, I/O operations, computations and other types of data manipulation are not in the text domain, we observe that, using the text opcodes intelligently, one can simulate other non-available opcodes. For example, although the opcode *move* is not available, one can easily simulate *move eax, ebx* by doing *push ebx* followed by *pop eax*. Moreover, as we will show next, we can dynamically create binary instructions once text-based malware sneaks into a computer system and gets a chance to be executed.

3.1.3 Construction of Text-based Malware

We use an ASCII worm to showcase the text-based malware. Suppose the ASCII worm passes through an ASCII filter to reach an email server. It exploits a buffer-overflow vulnerability on the execution stack of a running program and subsequently gains the control for execution. Since the opcodes for system-level instructions, which are

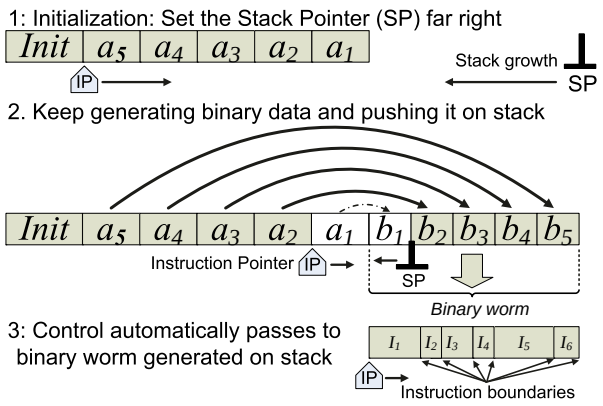

Figure 3-1. Creation of a binary worm on stack from text code

essential for a potent worm, are absent in the text domain, the only way to make them available is for the text worm to create them on the fly, preferably on the stack where the buffer-overflow attack has just transferred the CPU execution from the exploited program to the worm.

The method [56] for the text worm to generate binary instructions is illustrated in Figure 3-1. Denote the binary instructions to be generated as $B = b_1 b_2 ... b_n$. Let a_i be the text code segment that creates the binary word b_i. To give an example, starting with a word of four arbitrary text characters, a sequence of *inc* or *dec* instructions can produce a word of any value, though occasionally we can use *xor* and *sub* to do it more intelligently. We exploit the property that the Instruction Pointer *IP* and Stack Pointer *SP* move in opposite directions during the execution of stack-growth (push) instructions – *IP* increases while *SP* decreases, as shown in Figure 3-1. The code of the text worm is arranged in the order of a_n, a_{n-1}, ..., a_1. During execution, a_n generates b_n and pushes it on the stack, then a_{n-1} generates b_{n-1} and pushes it on the stack, ..., and finally a_1 generates b_1 and pushes it on the stack. The stack pointer must be appropriately set such that b_1 locates right next to a_1, which means that after executing a_1, the control will automatically pass on to the created binary instructions that begin from b_1. Hence, typical text malware

looks like $Ia_n a_{n-1}...a_3 a_2 a_1$, where I denotes some initialization code which performs bootstrapping such as setting the stack pointer to a proper position.

In this dissertation, we refer to the process of turning text malware into binary code as *decryption*. The malware itself must carry a *decrypter*, a cleartext ASCII instruction sequence that performs the decryption. In many cases, the whole malware is a decrypter, as shown in Figure 3-1. The reason for having such a long "hardcoded" $O(n)$-size decrypter for generating an n-word binary code will be apparent in Section 3.2.2.

3.2 Detection Strategy for Text Malware

In this section, we first show that the existing malware detectors are not adequately suited for detecting text malware. We then argue that MEL can serve as a differentiator between benign text and malicious text. Finally, we show that the MEL method, while applicable to text malware, can no longer apply to binary malware.

3.2.1 Limitation of Existing Binary Detectors

Since text is a subset of binary, one may be tempted to assume that binary malware detectors would be equally effective in detecting text malware. However, as we demonstrate through several research malware detectors as well as commercial ones, that assumption is not true.

Disassembly-based Detectors: Nearly a third of the instructions available in text are branch (jump) instructions. This high frequency of branch instructions is one reason why the disassembly-based binary malware detectors do not work well for text input. Since every branch forks an execution path into two directions, having too many of them increases the number of execution paths to be searched exponentially. So, unless some text-specific criterion is used to prune this search space, detectors may take excessively long time to run for text input. SigFree [73], which uses disassembly-based techniques, reported that it usually does not process text input due to performance degradation. This finding is corroborated by the observation that when we emulated

another disassembly-based detector APE [67] for text input (see table 3-2 in Section 3.6), the runtime was hours in many cases, clearly unacceptable for a malware detector.

Frequency-based Detectors: Detectors that rely on the statistical properties of the traffic to decide whether it is malicious are not foolproof against text malware either, as Koleshnikov *et al* [32] showed how an ASCII worm could easily evade a powerful and robust detector like PAYL [72].

Commercial Detector: We passed the text malware used in our experiments to the commercial malware detector McAfee, and no alarm was raised.

Summarizing the above discussion, we need a new detection strategy that can be efficiently applied to text input and will differentiate malicious text from benign text. Our strategy is based on the following observation: text malware has a high probability of having a high MEL, while benign text tends to have low MEL.

3.2.2 Text-Based Malware Has High MEL

Below are the two primary reasons why a text-based malware is likely to have a high MEL.

Opcode Unavailability: In order to be potent, a malware must perform certain actions, such as making system calls, which require opcodes that are unavailable in text data. Therefore, we see that text malware are constrained to *generate* these instructions dynamically. Since a binary malware usually has many such non-text instructions, dynamic generation of them entails long stretch of valid text instructions, which means high MEL.

Difficulty in Encryption: There are two difficulties associated with encrypting binary in text (and decrypting text back to binary). First, we observe that since the text domain is a proper subset of the binary domain, we cannot have a *one-to-one* correspondence between the two. Therefore, if we are encrypting one byte of binary data into text, the size of the encrypted output will be definitely more than one byte, which means not only the size of the encrypted payload will increase but also the decryption

logic will be more complex, thus resulting in a much larger decrypter. Second, in order to have a small decrypter routine, one needs to use a jump instruction with a negative displacement to "go back" to the beginning of the decrypter so that the same routine can repeatedly executed for decrypting different encrypted bytes. However, since all text bytes have 0 in their most significant bit (MSB), one cannot have a negative displacement in text – which means that all jumps in text instruction stream are in the forward direction. This precludes the possibility of having a small decrypter – for a n-word encrypted payload, one must have $O(n)$ decrypter blocks where each decrypter block will decrypt one individual word. Thus, we posit that text decrypters are large in size, and accordingly a text malware that employs encryption has a high MEL. While it is theoretically possible to overcome these difficulties (by generating the negative displacement dynamically or by using multi-level encryption), that would most likely make the decrypter more complicated and thus increase its size and MEL. We will discuss the multilevel encryption issue in greater detail in Section 3.8.

3.2.3 Benign Text Tend to Have Smaller MEL

A high MEL implies the presence of a long valid (error-free) instruction sequence. Therefore, if invalid (error-raising) instructions are dispersed abundantly in an instruction stream, the chances of having a high MEL is minimal. It transpires that the preceding is true for normal text stream – such invalid instructions do occur frequently in the benign text data due to the following reasons:

Prevalence of Privileged Instructions: The characters '*l*', '*m*', '*n*' and '*o*', which occur frequently in text [45], correspond to privileged I/O instructions that cannot be invoked from any user-level application without generating an error. Benign text data may have these instructions, whereas malware will never have them in its execution path.

Illegal Memory Access: Text instruction streams are very prone to segmentation fault due to attempts to access out-of-bound memory. This is because in Intel architecture, approximately two-third of all dual-operand instructions available in text (*xor*, *and*, *sub*,

and *cmp*) involve memory access. To see why, we note that unlike other architectures like VAX, Intel architecture does not allow two-operand instructions where both the source and destination operands are memory; however, it does allow register-register, register-immediate and register-memory instructions. Now, in order to have a register-register instruction in Intel architecture, the ModR/M byte of the instruction must have 1 in its Most Significant Bit, which is not possible in text. Therefore, for all two-operand instructions involving a ModR/M byte, one operand must come from memory for text instructions. This implies that with text, a significant proportion of the instructions involves memory access. With such abundance of memory-accessing instructions, while accessing the memory, a violation can happen in the following ways:

- If a register used for addressing the memory is uninitialized and thus contains an arbitrary value, then there is a significant probability that upon execution the memory addressed would be out-of-bounds, thereby causing errors.
- Considering that text domain includes all the possible segment override prefixes, and characters denoting those prefixes ('*d*', '*e*', '.', '6', '&' and '>') are frequent, there is a high possibility that a benign stream will have a Segment Selector prefix to the instruction that will cause it to access arbitrary memory segment and thus cause errors.
- When the memory address is explicitly stated, i.e. no register is used to point to a memory area, it can potentially create problems. This is because nowadays it is a common practice to randomize the starting addresses of user stacks and static libraries [4], and such explicit memory addresses might be out-of-bound for the program.

Thus, in a random (benign) text stream, such memory-accessing-error events are frequent.

3.2.4 Using MEL as Detection Strategy

Based on the discussion above, it is evident that unlike benign text, a text malware has a high probability of having a high MEL. Therefore, a threshold on the MEL can be used to determine whether a text stream is most likely malicious or benign. However, it should be pointed out that our contribution *not* limited to this rather straightforward approach of checking against an MEL threshold during pseudo-execution (which has appeared in previous works [67, 2] for classical binary worms), but rather 1) the discovery

of new text-specific techniques for identifying invalid instructions in order to prune the number of possible execution paths to be explored and demonstrating how adverse the detection results can be if we do not use those techniques, and 2) providing the probabilistic model of the MEL theory, which enables us to calculate the MEL threshold mathematically rather than empirically, and predict what would be the false positive rate for a user-specified MEL threshold.

We also show that the MEL method, though used for detecting binary worms previously, cannot be used any more. This is because binary malware does not suffer from the same encryption difficulties as the text malware does. Without any constraint on encryption, it is fairly easy for the binary malware to use a very short decrypter, which will result in a low MEL similar to random (benign) binary stream, thereby making the malicious traffic virtually indistinguishable from the benign, from an MEL-based detection purpose. Therefore, it is rather surprising that the MEL method was ever successfully used for detecting binary worm, as in APE [67] and Stride [2]. The reason those detection methods succeeded is because those schemes exploited a special property of the binary worms, viz. the fact that binary worms were accompanied by a NOP sled. Those schemes were directed not towards detecting the actual payload (which could be encrypted and thus have a small MEL) but towards detecting the worm's sled (a long sequence of unencrypted valid instructions, and thus having a high MEL). Unfortunately, according a recent survey [20], NOP sleds are almost never used nowadays, probably because the stack addresses today can vary by millions of bytes, and having a sled that long is improbable. Nowadays, most of the worms rather use the "register spring" method that involves no sled [20]. Thus, MEL-based methods (including APE or Stride) are not suitable for today's binary worms any more.

3.3 Probabilistic Analysis of MEL

In this section, we answer this question: given a sequence of n instructions with each instruction having a probability p of generating an error during execution, what is the

distribution of the longest error-free execution path (MEL)? We elaborate below why deriving the distribution of MEL is important.

We use the detection strategy that, if an incoming instruction stream contains a contiguously valid instruction sequence longer than a certain threshold τ, then it contains a malware with a certain *false-positive* probability α (which is the chance for a benign stream to have a contiguously valid instruction stream of length more than τ purely by accident). It is intuitive to see that the larger the value of τ is, the smaller the value of α will be. Unfortunately, if we aim at driving α close to 0 in order to avoid false alarms altogether, τ will be very large, which may lead to *false negatives* (the case that real malware is not detected). Therefore, it is important to characterize the trade-off between false positive and false negative by deriving the mathematical relationship between α and τ. Such a formula will allow the user to select a specific combination of the two values in order to achieve certain desirable performance. While it is possible to estimate the relationship between α and τ experimentally through a training data set, such data can be biased, not representing the general case. In this section we take a probabilistic approach that correlates τ with α using the character frequency distribution of text input.

3.3.1 Description of the Model for MEL

In our probabilistic analysis, we use Bernoulli trials to model this problem. We start with the assumption that the instructions in a normal input stream occur randomly and independently (this assumption is verified in Section 3.3.3). Let I_v denote a valid instruction, I_{inv} an invalid instruction, and p the probability for an arbitrary instruction disassembled from a normal stream to be invalid. Consequently, the probability for an instruction to be valid is $(1-p)$. Let n be the number of instructions in an input stream and N be the number of invalid instructions in the stream. It is easy to see that there are $N+1$ contiguously valid instruction sequences, each containing zero or more I_vs and terminating with an I_{inv}. The instruction sequence after the last I_{inv} does not have the terminating I_{inv}. For example, with $n = 17$ and $N = 5$, the following instruction

stream, $\overline{I_v I_v I_{inv}}\ \overline{I_v I_v I_v I_v I_{inv}}\ \overline{I_v I_v I_v I_{inv}}\ \overline{I_v I_v I_{inv}}\ \overline{I_{inv}}\ \overline{I_v}$, contains 6 such sequences (instructions in the same sequence are under the same bar), where the longest valid instruction sequence is $I_v I_v I_v I_v I_{inv}$ (MEL=5). Now, if we use the term X_i to denote the length of each of these $N+1$ valid instruction sequences, then the MEL is given by $X_{max} = \max\{X_1, X_2, ..., X_{N+1}\}$. Each X_i follows Geometric distribution with parameter p. Although $\sum_0^{N+1} X_i = n$, the X_is can be assumed to be independent (effect of this approximation will be discussed in Section 3.3.3). Below we derive Prob$[X_{max} \leq x]$, $\forall x \in [0..n]$, which is the cumulative density function of X_{max} (or the MEL).

First, we derive the conditional probability when the number of invalid instructions is fixed at $N = k$, for a specific number k (we would generalize it later).

$$
\begin{aligned}
&\text{Prob}[X_{max} \leq x \mid N = k] \\
&= \text{Prob}\,[max(X_1, X_2, X_3, ..., X_{k+1}) \leq x] \\
&= \text{Prob}\,[(X_1 \leq x) \text{ and } (X_2 \leq x) \text{ ...and } (X_{k+1} \leq x)\,] \\
&= \text{Prob}\,(X_1 \leq x) \times ... \times \text{Prob}\,(X_{k+1} \leq x) \\
&= [1 - (1-p)^x] \times [1 - (1-p)^x] ... \times [1 - (1-p)^x] \\
&= [1 - (1-p)^x]^{k+1}
\end{aligned}
$$

We stress that the probability calculated above is conditional on a specific value of N. The actual value of N may vary from 0 to n. Since N denotes the number of invalid instructions occurring among n instructions (with each of the n instructions having a probability p of being invalid), it follows the Binomial distribution with parameters (n, p).

Thus, Prob$[X_{max} \leq x]$ over all possible values of N is

$$\begin{aligned}
&\text{Prob}[X_{max} \leq x] \\
&= \sum_{k=0}^{n} \text{Prob}[N = k] \times \text{Prob}[X_{max} \leq x \mid N = k] \\
&= \sum_{k=0}^{n} \binom{n}{N} p^k (1-p)^{n-k} \times [1 - (1-p)^x]^{k+1} \\
&= (1 - (1-p)^x)[1 - p(1-p)^x]^n
\end{aligned}$$

The Probability Mass Function (PMF) for MEL is Prob$[X_{max} = x]$ = Prob$[X_{max} \leq x]$ − Prob$[X_{max} \leq x-1] = (1-(1-p)^x)[1-p(1-p)^x]^n - (1-(1-p)^{x-1})[1-p(1-p)^{x-1}]^n$.

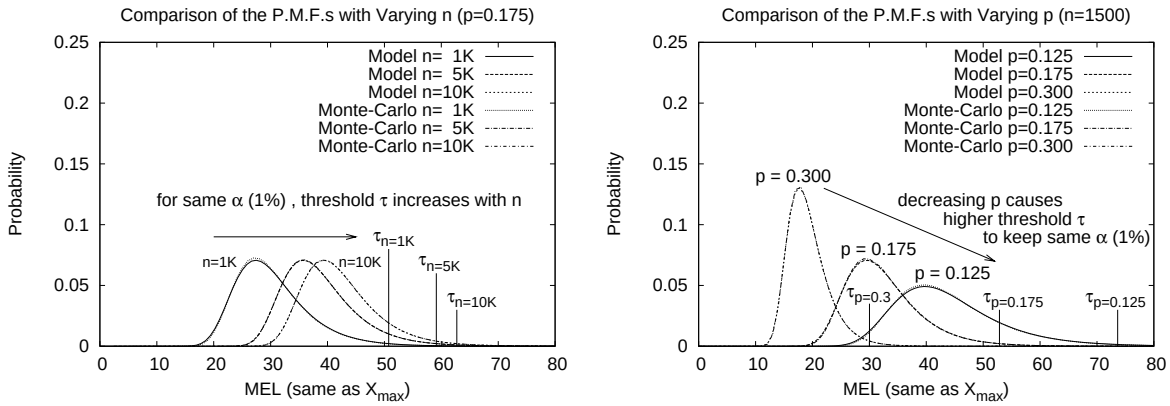

Figure 3-2. Juxtaposition of the PMFs for the MEL from the probabilistic model and from the Monte-Carlo simulation by varying n and p. A near-perfect match can be observed in almost all the cases.

3.3.2 Automatic Derivation of Threshold τ

We now derive the formal relation between τ and α. The resulting formula allows us to automatically derive the threshold value τ under the constraint that the false-positive probability is bounded by a given value of α.

False positive happens when X_{max} is greater than the MEL threshold τ. Thus, the false-positive probability must be α = Prob$[X_{max} > \tau]$ = $1 - \text{Prob}[X_{max} \leq \tau]$ = $1 - (1-(1-p)^\tau)[1-p(1-p)^\tau]^n$. We can approximate it as $\alpha = 1 - [1-p(1-p)^\tau]^n$ since $(1-(1-p)^\tau) \approx 1$. Thus, we obtain $\tau = \frac{\log(1-(1-\alpha)^{\frac{1}{n}}) - \log p}{\log(1-p)}$. This formula implies

that given n and p, we can calculate the MEL threshold τ corresponding to any allowable false-positive rate α chosen by the user. How to determine the values of n and p will be discussed in Section 3.5.2.

To verify that the above approximation has insignificant impact on the value of τ, we compare the values of τ obtained using the formula with or without the approximation. For example, when $\alpha = 1\%$, $n = 1540$ and $p = 0.227$ (the parameters used in our experiments), $\tau = 40.61$ with the approximation and 40.62 without (difference of 0.02%). Other reasonable parameter settings also show that the approximation induces only small error in the computation.

Based on the above analysis, if we use the derived τ as threshold, the false-positive probability will be bounded by α. This is very important, since it gives us the flexibility to set the detection sensitivity of an MEL-based detector.

3.3.3 Verification of the MEL Model

In our model, we assume that valid and invalid instructions occur independently in the benign text. If we can show that the validity of an instruction is independent of the validity of the instruction prior to that, then by induction it can be shown that occurrence of any valid or invalid instruction in an instruction stream is an independent event. To prove that, we conduct the Pearson's χ^2 test with the null hypothesis H_0 : in a pair of contiguous instructions $< I_1, I_2 >$, the validity of I_2 is independent of the validity of I_1. To verify this, we construct below the 2×2 contingency table of frequencies as follows. First we disassemble the benign text data used for our testing. Then, considering all possible contiguous pairs of instructions, we count the total number of cases for each of the 4 possible validity combinations and tabulate the results under the "observed" column. The rightmost columns under "expected" indicate the expected numbers as per Pearson's χ^2 test. As we observe, the values are very close; and the corresponding p-value (0.1) is not statistically significant to reject H_0.

	Observed		Expected	
	Valid I_2	Invalid I_2	Valid I_2	Invalid I_2
Valid I_1	8960	2797	8922	2835
Invalid I_1	2797	938	2835	900

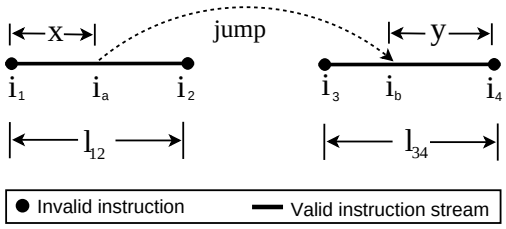

Figure 3-3. Effect of *jump* instructions

The other assumption in our model is that we do not enforce the condition that $\sum_0^{N+1} X_i = n$ and rather assume X_is occur independently. Is it evident that as n increases, the effect of this constraint becomes less pronounced. To verify this, we run Monte-Carlo simulation for the $PMF(X_{max})$ for different values of n and p. There, we toss a coin (with probability of head p) n times and calculate the MEL by taking the maximum distances between two heads that are separated by only tails and no heads in between. As heads are equivalent of invalid instructions, the maximum inter-head distance represents the MEL. The same experiment is run for thousands of rounds to obtain the distribution of MEL. Finally, we juxtapose the output *PMF* for the MEL from the Monte-Carlo simulation with the *PMF* generated by our probabilistic calculation in Figure 3-2. We observe a near-perfect match in all cases (especially with larger n), which vindicates our probabilistic model.

We also get one very important intuition from Figure 3-2. We see that if p decreases, it will require a higher threshold τ to keep the same false positive rate of α. However, higher threshold will mean that a lot of malware will also not get detected. Thus, to have a low value of false negative (in addition to a fixed low value of false positive α), we must find ways to increase p. This is why finding more ways to invalidate instructions in text streams is important.

3.3.4 Handling *Jump* Instructions in the Model

One minor issue with our probabilistic model is that it does not quite capture the effect of the *jump* instructions. However, as we will show below, the anomaly introduced by the jumps are minimal.

In our model, we had implicitly assumed that if instruction i_b is executed right after instruction i_a, then i_a and i_b must be contiguous in the original instruction stream. However, if i_a happens to be a jump instruction with i_b being the jump target, then we can be fairly certain that i_a and i_b will not be immediate neighbors in the disassembled instruction stream. This phenomenon is illustrated in Fig. 3-3, where solid dots represents invalid instructions (i_1, i_2, i_3 and i_4), and the straight line segments between the solid dots indicate stream of valid instructions. The length of a line segment (l_{12} and l_{34} in Fig. 3-3) indicates the number of valid instructions within that segment. Since we have observed that invalid instructions can be assumed to occur independently with probability of p, the length of the individual line segments is a random variable following Geometric distribution, with mean of $\frac{1}{p}$. Therefore, $E[l_{12}] = E[l_{34}] = \frac{1}{p}$. Now, suppose during runtime first x number of valid instructions are executed with the last one being the jump instruction i_a, and then another y number of valid instructions are executed, starting with the jump target instruction i_b and ending with the invalid instruction i_4. Let's assume i_a is located within the line segment between invalid instructions i_1 and i_2; and i_b is located within the line segment between invalid instructions i_3 and i_4. Thus, $x+y$ will be the total number of instructions in a possible execution path. Now, if the expected value of $x+y$ is any different from $\frac{1}{p}$, which is the expected value of such an execution path in our original model, then we have a problem. To illustrate the point, if i_a and i_b happen to be closer to i_1 and i_4 than i_2 and i_3 respectively, then the value of $x+y$ will be significantly smaller than $\frac{1}{p}$. Conversely, if i_a and i_b are closer to i_2 and i_3 than i_1 and i_4 respectively, then the value of $x+y$ will be significantly larger than $\frac{1}{p}$. However, since we are discussing a probabilistic event, a single instance of a large or small value does not matter – what

we are interested is studying whether having a jump instruction introduces a *bias* in the system by altering the *expected* value.

We investigate this problem in the following way. Given that a jump instruction occurs during the execution of a valid instruction stream, the location for the jump instruction is random, i.e. it can happen anywhere in that instruction stream. Pictorially, the location of jump instruction i_a is random between i_1 and i_2 in Fig. 3-3. Extending the same logic for the location of the jump target, we find that the location of i_b is also random between i_3 and i_4. Therefore, x and y can be perceived as two random variables, having the ranges of values between 1 and l_{12} for x and between 1 and l_{34} for y. Therefore, each of the random variables x and y will follow a discrete uniform distribution, with the following probability distribution (conditional on l_{12} and l_{34}):

$\text{Prob}[x = k] = \frac{1}{l_{12}} \quad \forall\ k = 1...l_{12}$, and

$\text{Prob}[y = k] = \frac{1}{l_{34}} \quad \forall\ k = 1...l_{34}$.

Thus, conditional on the given values of l_{12} and l_{34},

$E[x] = \frac{l_{12}}{2}$ and $E[y] = \frac{l_{34}}{2}$.

Since the displacement in a jump instruction is completely arbitrary, the position of i_a within the range $[i_1, i_2]$ does not affect the position of i_b within the range $[i_3, i_4]$. Thus, the random variables x and y are independently distributed. Therefore, the expected number of consecutively-executed valid instructions involving a jump instruction = $E[x + y] = E[x] + E[y] = \frac{l_{12}}{2} + \frac{l_{34}}{2}$ conditional on the lengths of the line segments being l_{12} and l_{34}. When we remove the condition and calculate the unconditional expected value, $E[x + y] = E[\frac{l_{12}}{2} + \frac{l_{34}}{2}] = \frac{\frac{1}{p}}{2} + \frac{\frac{1}{p}}{2} = \frac{1}{p}$, which is the same value of the expected number of valid instructions executed between two invalid instructions in our original model without considering the jump instructions.

Therefore, we observe that having a jump instruction among a stream of valid instructions does not affect the distance between the invalid instructions in a probabilistic sense. This implies that the MEL from our model, which originally did not account for the

jump instructions, is affected by it only in a very minor way. The rationale behind this inference is as follows. Due to presence of a jump instruction, the only parameter that is directly affected is n, since a jump essentially discards the instructions located between the jump instruction and the jump target, thereby lowering n. With a smaller n, the threshold τ corresponding to an error rate α would be marginally smaller than the original scheme. Therefore, 100% of the alerts that are generated by our model without considering the jump would have also been generated by a model that considers it, which means the false positive rate is not affected. The only downside is that now we will be missing a few alerts that should have been raised, which means the false negative value would increase slightly. However, as we know from the experiences of anti-malware product vendors, false positive is a much bigger problem than false negative, since the benign traffic volume dwarfs the malicious. Therefore, we argue that our model is not significantly affected by the presence of jump instructions.

3.4 Implementation of DAWN

This section describes DAWN, the detection strategy for ASCII worms. Briefly, DAWN operates in two main stages: instruction disassembly and instruction sequence analysis. First it disassembles the ASCII input from every possible position, and attempts to see if any such disassembly could potentially lead to a malicious code by performing pseudo-execution of every possible path (details of this "pseudo-execution" would be provided later). If it detects a long sequence of valid instructions (longer than a certain threshold), an alarm is raised. The individual steps are delineated in more detail in the next two subsections, followed by the experimental results.

3.4.1 Step 1: Instruction Disassembly

Since it is not possible to predict the entry point of the worm in the input stream, the input (say of length n bytes) needs to be disassembled from all possible n entry points. It has been shown [9] that due to the self-adjusting nature of the Intel instructions, if one starts interpreting the same instruction stream from two adjoining bytes, the instruction

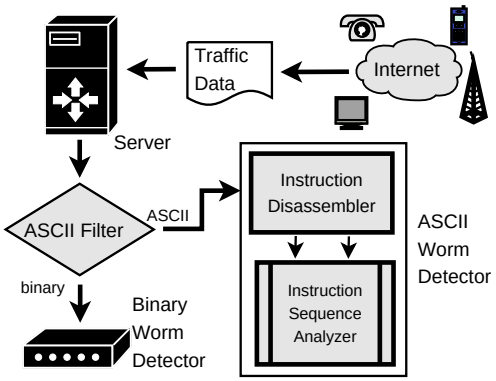

Figure 3-4. How DAWN works

boundaries of the two instruction sequences tend to get aligned within 6 instructions (max 78 bytes) with a very high probability. Thus, for every entry point we need to disassemble only an average of 6 instruction before we can re-use the instruction sequence that has already been disassembled. Therefore, although the disassembly is technically a $O(n^2)$ process, it is linear from a practical standpoint.

3.4.2 Step 2: Instruction Sequence Analysis

The main purpose of this stage is to ascertain how long an instruction sequence (which may start anywhere within the text data) may execute without generating an error. The error may result either from using a privileged instruction, or from a memory access violation. As DAWN proceeds with the pseudo-execution of the instruction sequence, it keeps track of which registers have been initialized properly. When an uninitialized register is used to address memory (as source or destination), it is considered to be the end of that sequence. For a control flow bifurcation (like *jump*), DAWN recursively considers both the possible routes (*jump target* as well as the *fall-through* instructions) and chooses the longest path between the two. It should be noted that as there are only forward jumps in text data, there is no chance of DAWN "looping around" in the code endlessly. If the length of the longest executable instruction sequence exceeds a certain threshold (considering all n possible entry points), then an alarm is raised.

The sketch of the detection algorithm implementing the above ideas is given below.

Algorithm 1 DetecttextMalware (printable ASCII stream A)

```
1: D = disassembleInstructionsFromEveryEntryPoint(A);
2: for startPoint s = 1 to size(A) do
3:    v ⇐ 0;      // max length of valid instructions till now
4:    Π ⇐ ∅;      // set of properly populated registers
5:    RecursiveDetect(D, s, v, Π);
6:    if v > threshold then
7:       Raise Worm Alert;
8:    end if
9: end for
```

Algorithm 2 RecursiveDetect(disassembled instructions D, entry point s, max valid length v, populated register set Π)

```
1: i_s ⇐ D[s];      // Instruction starting at byte s
2: s_next ⇐ s + length(i_s);      // location of the next instruction
3: if i_s is truncated then
4:    return;
5: end if
6: v ⇐ v + 1;      // Increase the MEL counter
7: if i_s is a privileged instruction or accesses memory with an inappropriate segment override
   prefix then
8:    return;
9: else if i_s is a single-register or register→stack instruction (e.g. inc, dec, push) then
10:   RecursiveDetect(D, s_next, v, Π);
11: else if i_s is a immediate→register or stack→register instruction (e.g. pop, popa) then
12:   Π ⇐ Π ∪ (destination registers);      // Initialization
13:   RecursiveDetect(D, s_next, v, Π);
14: else if i_s is a register–memory operand instruction (e.g. xor) then
15:   Σ ⇐ memory-accessing registers;
16:   if Σ ⊈ Π then
17:      return;
18:   else
19:      Π ⇐ Π ∪ (destination registers);      // Initialization
20:      RecursiveDetect(D, s_next, v, Π);
21:   end if
22: else if i_s is control-flow instruction (e.g. jne, jae etc.) then
23:   v_target ⇐ v_fallthrough ⇐ v;
24:   Π_target ⇐ Π_fallthrough ⇐ Π;
25:   RecursiveDetect(D, s_target, v_target, Π_target);
26:   RecursiveDetect(D, s_fallthrough, v_fallthrough, Π_fallthrough);
27:   v ⇐ max(v_target, v_fallthrough);   // Set Π accordingly
28: end if
29: return;
```

3.5 Evaluation

We evaluate the effectiveness of our MEL theory in the following steps: (i) Creating the test data, (ii) Determining the appropriate threshold from the created test data using the MEL theory, (iii) Running the detection algorithm (DAWN [41]) with the threshold determined in the previous step, and observing the false positive and false negative rates. The tests were run on an Intel(R) Pentium-IV 2.40 GHz CPU with 1 GB of RAM in a Linux machine.

3.5.1 Creation of the Test Data

For creating the text malware, the frameworks provided by Rix [56] and Eller [26] were used to convert multiple binary buffer overflow programs (from [3]) into their text counterparts and more than one hundred text worms were created in that way. The effectiveness of each text malware was tested by actually running the vulnerable program and then by observing the spawning of the shell. To check whether a text malware detector is at all needed, McAfee antivirus program was run on both the binary and text shellcodes and it raised alarms for the binary cases only. For creating the benign dataset, approximately 500 KB of real web traffic from our departmental network were collected using Ethereal. After stripping off the headers, 100 cases, each containing approximately 4K text characters, were selected to serve as the benign data.

3.5.2 Determining MEL Threshold τ

Since we can express the threshold τ as a function of the false-positive probability α with parameters p (probability of an invalid instruction) and n (total number of instructions), we need to first calculate the values of these parameters (p and n) for our test data. For the calculations, we use only the following two entities: 1) the input size C (in number of characters), and 2) the character frequency table (indicating the probability of occurrence for each character), which can either be pre-set (from experience) or can be obtained by a linear sweep of the input character stream in case no pre-set data exists (like our test condition). We do not need to disassemble any data for determining p or n.

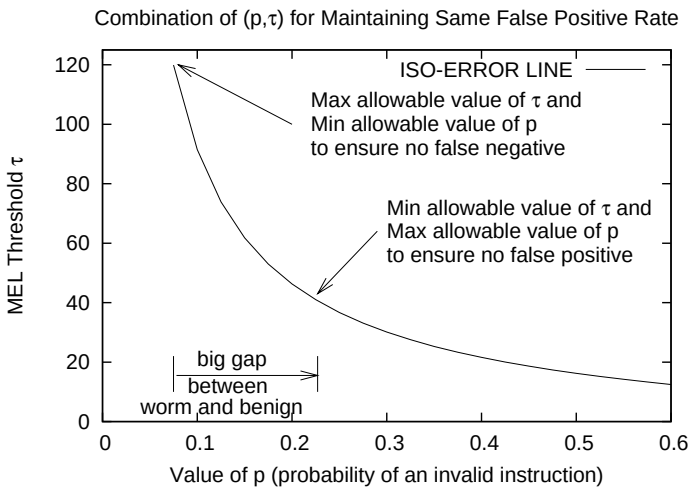

Figure 3-5. Correlation between τ and p for maintaining same error (false positive) rate $\alpha = 1\%$.

Determining n: We know that the total number of instructions $n = \frac{C}{\text{Avg Instr size (bytes)}}$. The average length of an instruction is given by E[length of instruction] = E[length of prefix chain] + E[length of *actual* instruction]. By *actual* instruction, we denote the rest of the instruction *after* the prefix chain, starting with the instruction opcode and including ModR/M, Immediate, Displacement, SIB, etc. Now, if z denotes the probability that a character is one of the instruction-prefix characters ($z = 0.16$ in our case), then E[length of prefix chain] = $\sum_{i=0}^{\infty} i \times \text{Prob}[\text{length(prefix chain)} = i] = \frac{z}{(1-z)} = 0.19$. Similarly, E[length of actual instruction] = $\sum_i \text{length}[\text{instr}(i)] \times \text{Prob}[\text{instr}(i)]$, where instr($i$) represents an *actual* text instruction. In our case, E[length of actual instruction] was found to be 2.4. Thus, E[length of instruction] turns out to be $0.19 + 2.4 \approx 2.6$ bytes per instruction. Since $C = 4$K in our case, $n = \frac{C}{E[\text{instruction size}]} = \frac{4000}{2.6} \approx 1540$.

Determining p: We obtain p by adding the probability of I/O instructions and wrong-Segment-override memory-accessing instructions (which are 18.5% and 4.2% according to the frequency distribution of our test data). We disregard the probability of illegal memory access due to unpopulated memory-addressing register, as it requires evaluation of prior instructions and hence it cannot be determined standalone whether it will cause an abort or not. We also take the conservative approach of not using the

possible error due to explicit memory address, as the register spring technique exposes the usage of static addresses in Windows [20] for some cases. Thus, p turns out to be $0.185 + 0.042 = 0.227$ in our case.

Determining the threshold τ: We set the false positive rate at $\alpha = 1\%$. For this α, we calculate the corresponding threshold $\tau = \frac{\log(1-(1-0.01)^{\frac{1}{1540}})-\log 0.227}{\log(1-0.227)} = 40$ for our calculated experimental parameters $n = 1540$ and $p = 0.227$.

3.5.3 Experimental Results

In our experiments, the MEL threshold of 40 catches all the malicious cases and not a single benign case gets misclassified as malicious, thus yielding zero false positive and zero false negative rates. To interpret the result even further, we take the MEL from each benign as well as malicious input data, and construct the overall MEL frequency charts (equivalent of PMF of MEL). We compare the frequency distribution of the MEL for benign and malicious test data in Figure 3-6. For the benign data, the average MEL is near 20, and max MEL is 40 (same as τ), which matches our expectations very well. On the other hand, for the malicious data, the minimum MEL is 120, thereby marking a clear differentiator. Also, if we connect the frequency points for benign, we observe that it forms into a shape somewhat similar to the PMF curves in Figure 3-2, which shows that our model is indeed mimicking the actual behavior. Also, we observe from Figure 3-5 that the gap between the false positive boundary (p value of 0.227 corresponding to MEL of 40) and false negative boundary (p value of 0.073 corresponding to MEL of 120) is quite large, which means even if the estimated p changed by a small margin, we would still have been able to distinguish the malware from the benign data. Also, the average instruction length from our actual experiment (2.65) was found to be very close to our expected value (2.6) assuming character and instruction independence.

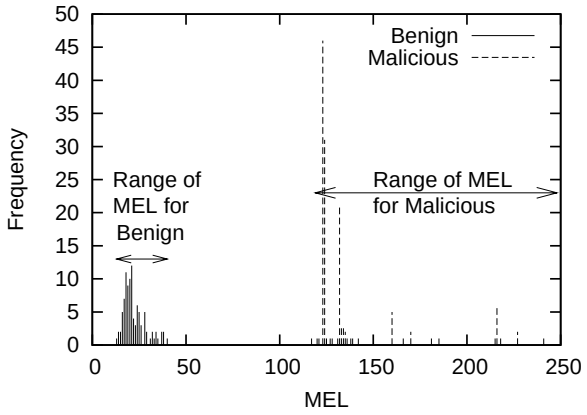

Figure 3-6. Comparison of MEL frequency charts for benign and malicious text traffic for DAWN

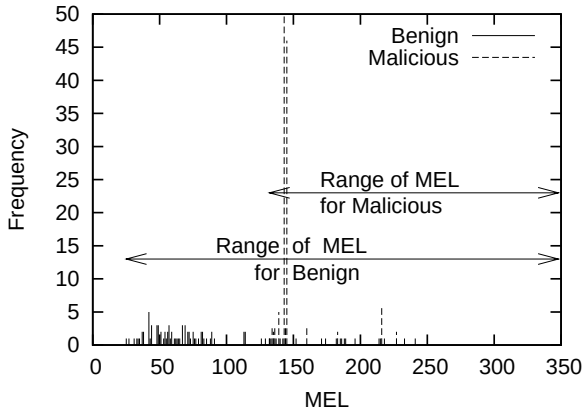

Figure 3-7. Comparison of MEL frequency charts for benign and malicious text traffic for APE-L

3.6 Comparing Our Work with Others

Since our detection mechanism is completely payload-based, we only compare our work with similar schemes. We contrast our method with 1) Abstract Payload Execution (APE [67]), and 2) SigFree [73].

3.6.1 Contrasting with APE

While our work generally follows the direction shown by APE [67] that introduced the concept of MEL, there are a number of significant differences:

- APE did not provide any mathematical foundation of the underlying model, which we do achieve here. As a result, any MEL thresholds in APE is obtained *experimentally*, while in our case the threshold is calculated automatically by the model – there is no "parameter tuning".
- APE runs on random samples of data, while we examine the full content.
- Unlike our malware detection strategy, APE is not a malware detector but a worm detector. It worked for worms because previously worms used to have a sled, a feature that is almost obsoleted now [20]. As a result, APE's effectiveness is severely dwindled today.

- Although text malware do have large MELs, we found that APE, in its proposed form, is not effective for detecting them either. This is because APE, which was designed for binary worms, did not exploit the text-specific properties. The definition of invalid instruction there is narrower than ours; APE considered an instruction invalid only when it is either incorrect or has a memory operand accessing an illegal address. This is a special case of our definition; we introduce new ways to invalidate more instructions in text (like I/O instructions). Moreover, the APE paper [67] did not present *specific* methods to determine which instructions are valid and which are invalid.

To elaborate the last point, we implemented an APE-mimicking algorithm (calling it APE-L) that did not exploit the text-specific constraints discovered by us, and compared its detection sensitivity and runtime with DAWN's. The reason for us implementing it is the unavailability of an updated and working implementation of APE. As seen from the detection sensitivity comparison charts in table 3-1 and in figures 3-6 and 3-7, the range of MEL for malicious and benign is distinct for DAWN but overlapping for APE-L, which means the APE would not be suitable for differentiating malicious text from benign. The results of the comparison showed clearly that APE is ineffective for text. Also, as mentioned earlier, the high frequency of jump instructions in ASCII data causes the number of possible execution paths to increase exponentially. So, unless the ASCII-specific criteria is used to invalidate instructions to prune this search space, detectors may take very large time to run for ASCII data. This finding is corroborated by the observation that compared to DAWN, APE-L runs much slower for ASCII (table 3-2), to the extent that for some cases APE-L does not even terminate for hours.

Table 3-1. Comparison of DAWN and APE-L for detection sensitivity

Sensitivity	MEL Avg		MEL Range	
	DAWN	APE-L	DAWN	APE-L
Benign	22.5	73.7	13 – 46	25 – 359
Malicious	138.1	152.9	117 – 327	132 – 353

Table 3-2. Comparison of performance (runtime) for DAWN and APE-L

Performance	Runtime Avg		Runtime Range	
	DAWN	APE-L	DAWN	APE-L
Benign	0.58s	22.0s	$0-1$s	$0-3$hr
Malicious	0.23s	0.3s	$0-1$s	$0-2$s

3.6.2 Contrasting with SigFree

SigFree [73] is a zero-day buffer-overflow detector that detects the worm by counting the length of only the *useful* instructions in an instruction sequence, while our approach counts all executable instructions, irrespective of whether they are useful or not. While SigFree claims to have the capability to catch ASCII worms, it incurs significant computational overhead in order to examine the ASCII traffic, as a result of which it *usually* bypasses the ASCII traffic to enhance performance. On the other hand, in our case, processing the ASCII stream is very fast. Finally, one of the criteria for measuring the *usefulness* of an instruction in SigFree is that it must not have any data anomaly, as for example by checking if the sources have been properly populated or not. However, we show in the following example that it may be possible to make SigFree think that the data anomalies have happened while actually none happened. One of the data anomalies that SigFree reports are *undefine-reference*, which happens when a variable, which is not yet *defined* (*i.e.*, not populated properly), is *referenced* (used as source) again. SigFree posits that the state of an undefined variable remains undefined when its value is reset with an undefined value. However, we show that even by using the undefined variable as a source, one can properly define, i.e., initialize a variable. The following example would make it clear. Suppose the register variable `eax` initially contains junk value, which implies `eax` is in *undefined* state. Now if we do `and eax, 0x20202020`, followed by `and eax, 0x40404040`, according to SigFree `eax` will still remain in *undefined-reference* state since the source register referenced in this case, `eax`, was in *undefined* state. However, the two instructions mentioned above actually sets `eax` to zero irrespective of its previous content, which means it is possible to reach a *defined* (initialized) state even with an undefined

reference. In DAWN, we overcome this flaw by taking the conservative approach that any undefine-reference is also a potentially defined state, thus increasing the detection probability.

3.7 Text Malware in Other Architectures

In Section 3.1, we discussed the constraints and construction of a text malware in Intel 32-bit architecture (IA-32). We observed that one of the requirements for having text instructions is that the most significant bit (bit 7) of every byte in the instruction stream must be 0, using the notation that the eight bits of a byte are labeled as bit 0 through bit 7 (from the least significant to the most significant). Also, since the ASCII characters 0x00 through 0x1E are also unprintable, a printable character byte must have the bit 5 and/or bit 6 set. We get one important insight from these constraints: the shorter the size of the instruction, the more the possibility of finding instructions that are completely text. This is because if an instruction is x bytes long, then each of the x bytes must individually be text in order to make the whole instruction text, a requirement which is often difficult to meet. The advantage of IA-32 being a CISC (complex instruction set computers) architecture is that a significant number of instructions are only a byte or a few bytes long, and it is easy to find text instructions among those short instructions. However, when we investigate the RISC (reduced instruction set computers) architectures, we observe that each instruction there has a fixed, relatively longer width (mostly 4 bytes for 32-bit architectures). We suspect that with such comparatively longer instructions, it might be difficult to have adequate number of text instructions required to create a text malware. In this section, we explore two such RISC architectures (MIPS and SPARC) and discover evidences that corroborate our suspicion.

3.7.1 MIPS Architecture

A MIPS instruction is 4 bytes long (for MIPS version IV [52]). Fig. 3-8 depicts the 3 different families of instructions in MIPS (Register type, Immediate type and Jump type).

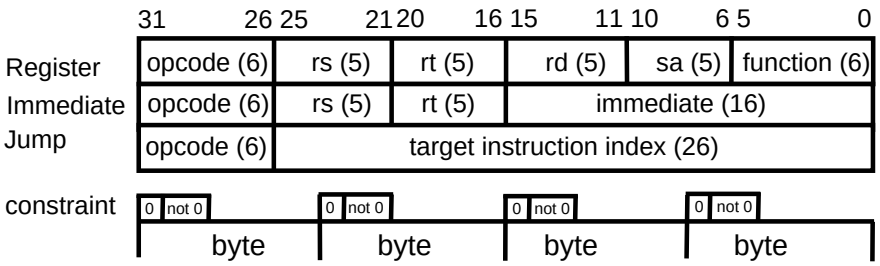

Figure 3-8. Decoding MIPS Instructions into different fields (with field lengths), along with text constraints and byte boundaries.

It also shows the requirements for the instruction to be text. The implications of this constraint are as follows:

- **Opcode:** In order to remain text, bit 31 must be 0 while at least one of bit 29 or bit 30 must be set. This rules out most of the available instructions, since the opcode is zero for most instructions (further decoding into different instructions is carried out by examining the function bits).
- **Source register *rs*:** Since this field crosses over any byte-boundaries, to remain text, bit 23 must be 0, and at least one of bits 21 and 22 must be 1. The last constraint eliminates those instructions that mandate a zeroed-out rs field, e.g. `LUI`.
- **Target register *rt*:** There are no restrictions imposed on this field.
- **Destination register *rd*:** To remain text, bit 15 must be 0, and bit 14 and/or bit 13 must be set. Therefore, only registers 4 through 15 are usable (thus excluding stack pointer, frame pointer, global area pointer, return address pointer etc.).
- **Shift amount *sa* and function:** Bit 7 of sa must be zero, and either the least significant bit for the sa field, or the most significant bit of the function field must be set (or both of them).
- **Immediate:** Since bits 13 and/or 14 must set (also the case for bits 5 and 6), no immediate value less than $2^{13} + 2^5$ can be used.
- **Target instruction index:** Again, due to similar reasoning for the immediate field, no address index value less than $2^{21} + 2^{13} + 2^5$ can be used.

After eliminating those instructions that do not satisfy the text constraints mentioned above, the only instructions that remain are ADDI, ADDIU, SLT, SLTIU, ANDI, ORI, XORI, LUI, BEQL, BNEL, BLEZL, BGTZL, COPz and JALX. Notably absent is SYSCALL, which is a must for any malware. While any binary byte can be recreated on the stack at runtime (just like IA-32), here we are constrained to do so using text instructions only. Since MIPS is a load-store architecture, the only way to modify the memory is through the store commands such as SB, SH, SW, SD, SWL, SWR, SDL, SDR, SC, SCD, SWCz, SDCz, SWXC1 and SDXC1, none of which are available in text. Therefore, we argue that unless future revisions of MIPS allow text-based STORE instructions, it will be very tricky to have a text shellcode in MIPS.

3.7.2 SPARC Architecture

SPARC is also a CISC architecture with fixed 32-bit wide instructions. According to SPARC V9 architecture manual [74], there are primarily 4 formats of instructions, based on the value of the "op" field (bits 30 and 31):

- **Format 1 ($op = 1$):** CALL instructions, with 30-bit displacement.
- **Format 2 ($op = 0$):** SETHI and branches (Bicc, BPcc, BPr, FBfcc, FBPfcc).
- **Format 3 ($op = 2$ or 3):** Arithmetic, Logical, MOVr, MEMBAR, Load, and Store.
- **Format 4 ($op = 2$):** MOVcc, FMOVr, FMOVcc, and Tcc.

We recall that similar to MIPS, in order for a SPARC instruction to be text, bits 7, 15, 23 and 31 must be unset, while at least one of the bits in each of the following bit combinations must be set: (5,6), (13,14), (21,22) and (29,30). Since the field *op* spans bits 30 and 31, instructions belonging to Format 3 and 4 categories are immediately ruled out since there the bit 31 must be set. Thus, effectively it leaves us with only CALL, SETHI and branch instructions, none of which can modify the memory. While CALL instructions can potentially divert the execution flow, we doubt whether that alone would suffice for an effective shellcode.

3.8 Limitations and Conclusions

In this section we discuss some of the limitations of our detection strategy. We reiterate the basic principle of our detection method: 1) A text malware must self-mutate to generate potent binary opcodes, 2) this mutation requires a significant number of memory-writing instructions, 3) due to difficulties in encryption (including unavailability of loops in text and lack of one-to-one correspondence between text and binary domains) the size of a decrypter is relatively big for text malware, 4) due to randomness property, benign text data does not have such a long error-free executable instruction sequence, and 5) the length of the maximal valid instruction sequence can thus be used to differentiate between benign and malicious text data. We have already mentioned that generating loops dynamically makes the decrypter complicated and thus longer. We discuss in detail a possible argument against the other encryption difficulty (lack of one-to-one correspondence) below.

We have argued that the absence of one-to-one correspondence between text and binary makes the task of decryption more complex and thus causes the decrypter to be large with high MEL. However, one may overcome this obstacle by using multilevel encryption (Russian doll architecture) in the following manner. First, convert the binary malware into text, and then encrypt this text malware in such a way that the output is yet again text. We observe that in the second step, we are doing encryption within the *same* text domain, which signals the possibility of having a one-to-one correspondence. On the surface, this approach appears to have merit since 1) the final encrypted text data will show very little trend of a text malware, and 2) because of the one-to-one correspondence, one may be able to use *simple* decryption schemes, which means a short decrypter. While it is impossible to consider all possible encryption methods, we put forth our rebuttal to this argument by demonstrating the case of using `xor`, which is usually a favorite choice for encryption. First of all, we observe that there is no *single* decryption key (a text byte) with the property that `xor`-ing it with any other text byte will still yield text

data. This is because the text data (0x20–0x7E) occupies a somewhat odd slot in the original ASCII table, and `xor`-ing two characters from text data often yields a result that is not text. As shown in Figure 3-9, after dividing the 95-char text domain into three nearly equal-sized parts (viz. 0x20–0x3F, 0x40–0x5F, and 0x60–0x7E), if we `xor` *any* two bytes from the *same* part, then the output will belong to the non-text domain 0x00-0x1F. This means that, in order to use `xor` directly, we cannot use a *constant* key for all of the text. Consequently, the decryption logic will have to be more complex too, leading to a not-so-small decrypter.

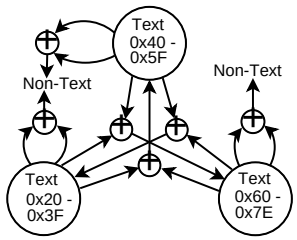

Figure 3-9. Encryption difficulties in using XOR for text

We emphasize that while we offer a novel way to differentiate between benign and malicious text traffic, this means that we have merely made the task of an attacker significantly harder. As per our limited experiment, the difference between the maximum length of the valid instruction sequence between benign and malicious traffic is currently significantly large. To the best of our knowledge, no text malware employing encryption to this date has been able to come up with a decrypter smaller than our current threshold. However, as security is a cat-and-mouse game, in future we will invariably see such malware, and we must strive to find more exploits to counter that.

We would like to reiterate that while our approach is similar to some other existing MEL-based schemes [67, 73, 2], a fundamental difference exists. In our detection scheme the MEL threshold is obtained purely from the statistical properties of text traffic, while for the rest it is obtained experimentally from the MEL of the benign data.

3.9 Contributions

We have analyzed the Maximum Executable Length method and laid mathematical foundation to this theory. Although the MEL method have been used by others, we are the first ones to show *why* it works. We have shown that such theory can be used

to detect malware in the text domain but no longer in the binary domain. We have incorporated our MEL model into a text malware detector that is easily deployable, signature-free, requires no parameter tuning, has user-configurable detection sensitivity, and is extremely robust.

CHAPTER 4
PROPAGATION MODELING OF THE PERMUTATION-SCANNING WORM

Worms have huge damage potential due to their ability to infect millions of computers in a very short period of time [42]. In order to counter that threat, we need to look into their content (for signatures) as well as propagation pattern (for Internet-scale behavior) [6, 53, 58, 37, 8, 38]. The propagation characteristics of a worm shows what kind of network traffic will be generated by that worm and how fast the response time must be for countermeasures. Therefore, in order to understand (and possibly counter) the damage potential of worms, it is very important to characterize their overall propagation properties.

Although modeling worm propagation has been an active research area [63, 80, 7, 75, 78], one might question the practical importance of such work if it is possible to obtain fairly good approximation of the worm's propagation characteristics by running a simulator for a sufficient number of times and taking the average. However, there are reasons why simulations may not always be able to produce the intended results. First, it often takes a long time, 16 hours in our case on a Intel Xeon 2.80GHz processor for 400M hosts that are estimated to be in today's IPv4 space, to simulate a single run of worm propagation for one set of worm/network parameters. To learn the average behavior, many such runs need to be performed, and the whole simulation process has to be redone for any parameter change, *e.g.*, for a different population size of vulnerable hosts or a different scanning speed of infected hosts. Second, the simulation overhead can be prohibitively high in some cases. Suppose we want to simulate a worm that exploits a commonly used Windows service on today's Internet. It means that the vulnerable population size could be in the order of several hundred millions as Windows machines predominate in the Internet. If there are 300M such computers, they will entail 300M records in the simulation, one for each vulnerable host. Even if each record is one integer (keeping its address alone), it will require a memory of 1.2 GB. Now, if we want to study

the impact of migration from IPv4 to IPv6 on worm propagation, a full-scale simulation of scanning the address space of size 2^{128} will be computationally infeasible. In comparison, numerical computation based on a mathematical model takes little time to produce the accurate propagation curves. Third, simulation results themselves do not always give the *mathematical insight* that a formal model provides. One may guess upon the impact of various parameters on worm propagation based on extensive simulations (which may take enormous time), but such guesses can never be as precise and comprehensive as an analytical model, which tells exactly why and by how much a parameter change will affect the outcome.

Traditionally, most modeling work [63, 80, 7, 75] concentrates on the relatively simple random-scanning worms, which scan the Internet either randomly or with bias towards local addresses in order to reach all vulnerable hosts. This strategy leaves a large footprint on the Internet (which reveals the worm's presence), and different infected hosts may end up scanning the same address repeatedly. In recent years, worm technologies have advanced rapidly to address these problems. By enabling close coordination among all infected hosts, the permutation-scanning worms (introduced in the seminal paper [63] by Staniford *et al.*) minimize the overall traffic volume for scanning the Internet through a divide-and-conquer approach. There, each active infected host is responsible for scanning a subset of all addresses, and this subset may vary over time. Such a cooperation strategy empowers the worm with the ability to propagate either much faster, or alternatively, much stealthier (if the infected hosts scan at lower rates). Warhol worms, which are similar to permutation-scanning worms with larger hitslists, have been shown using simulations to be able to infect the whole Internet in a matter of minutes [63]. However, understanding these potent worms through mathematical modeling has remained a challenge to date.

In this work, we propose a mathematical model that *precisely* characterizes the propagation patterns of the permutation-scanning worms. The analytical framework

captures the interactions among all infected hosts by a series of inter-dependent differential equations, which together describe the overall behavior of the worm. We then integrate these differential equations to obtain the closed-form solution for worm propagation. We use simulations to verify the numerical results from the model, and show how the model can be used to assess the impact of various worm/network parameters on the propagation of permutation-scanning worms. We also investigate the impact of dynamic network conditions on the correctness of the model, considering network congestion, bandwidth variability, Internet delay, host crash and patch.

In this work, we propose a mathematical model that *precisely* characterizes the propagation patterns of the permutation-scanning worms. The analytical framework captures the interactions among all infected hosts by a series of inter-dependent differential equations, which together describe the overall behavior of the worm. We then integrate these differential equations to obtain the closed-form solution for worm propagation. We use simulations to verify the numerical results from the model, and show how the model can be used to assess the impact of various worm/network parameters on the propagation of permutation-scanning worms. We also investigate the impact of dynamic network conditions on the correctness of the model, considering network congestion, bandwidth variability, Internet delay, host crash and patch.

The rest of this chapter is organized as follows. Section 4.1 describes the permutation-scanning worms. Section 4.2 introduces several important concepts underlying our mathematical model. Sections 4.3 and 4.4 present the exact propagation models for the basic permutation-scanning worm and its general extension, respectively. Section 4.5 derives the closed-form solution. Section 4.6 and Section 4.7 discuss how different worm/network parameters and real-life network constraints will affect the worm propagation, respectively. Section 4.8 draws the conclusion.

4.1 Anatomy of Permutation-Scanning Worms

We first describe the divide-and-conquer nature of the permutation-scanning worms. We then explain the reason for address permutation and discuss the stealthy potential of such worms.

4.1.1 Divide-and-Conquer

To avoid repeatedly scanning the same addresses, the infected hosts may collaborate by dividing the IPv4 address ring into disjoint sections, each of which will be scanned by one host. Each initially infected host starts "walking" along the address ring clockwise from its own location and *sequentially* scans the traversed addresses. Whenever it infects a host, it continues walking and scanning the addresses after that host, while the newly infected host performs a *jump*, i.e., chooses a random location on the ring and starts to walk and sequentially scan addresses clockwise after that location. The reason for this jumping action is that if the newly infected host instead started scanning sequentially after its own address, then those addresses would get scanned by both this host and its infector – a needless duplication of the same work. Now, if the scan performed by a host h_1 hits an already infected host h_2, h_1 knows that addresses after h_2 must have already been scanned by another infected host that infected h_2, or by h_2 itself in case h_2 was one of the initially infected hosts to begin with. In either case, it is unproductive for h_1 to continue scanning addresses after h_1. Therefore, h_1 *jumps* to a random location on the ring and starts to scan addresses clockwise after that location. An infected host retires (stops scanning) after hitting a certain number of already-infected hosts.

An alternative to the above random-jump approach is to assign each infected host an exclusive section of the address ring for scanning. As a host sequentially scans its section, when it infects another host, it assigns half of the remaining unscanned addresses to the latter and adjusts its section boundary accordingly. When a host reaches the end of its section, it retires. The problem with this approach is that it is not fault-tolerant. If one infected host is blocked out or somehow crashes, it may leave many addresses in its section

unscanned. Random jumps by infected hosts before they retire (as described previously) help solving this problem by providing redundancy, and this work will focus on such worms only.

4.1.2 Permutation

While the above divide-and-conquer method reduces the chance of scanning the same address again and again, it has a serious weakness. Since the IP addresses scanned by an infected host are contiguous, it is susceptible to be identified by address-scan detectors or other IDSs that look for worms performing local subnet scanning. To counter this, Staniford *et al* [63] shows that a worm can permute the IP address space into a virtual one (called the *permutation ring*) through encryption with a key. The divide-and-conquer method is then applied on this permutation ring. While each infected host logically goes through contiguous addresses on the permutation ring, it actually scans the IP addresses that the permuted addresses are decrypted to, which cannot be easily picked up by address-scan detectors because those IP addresses are pseudo-random and distributed all over the Internet.

4.1.3 Stealth

Fast propagation and stealth are two conflicting goals that the worm designers strive to balance. To spread fast, infected hosts should scan at high rates, which however makes them easier to be detected [53, 58, 42]. To be stealthy, they have to act as normal as possible by scanning the Internet at a controlled low rate, which is a worm parameter that can be set before release. A stealthy worm can be more harmful. A fast worm generates headline news, such as Slammer [42] that caused widespread network congestion across Asia, Europe and Americas. However, such a worm is more likely to be detected quickly and attract defense resources for its elimination. A stealthy worm propagates slower but may stay undetected for a long time, potentially doing more harm.

4.1.4 Hitlist

The initial part of worm propagation is most time-consuming, as only a few infected hosts perform scanning in a vast address space. Once the number of infected reaches a critical mass, the rate of new infections goes up drastically. To improve the initial scanning speed of a stealthy worm, one can use a *hitlist* as proposed in [63], which is a pre-compiled list of target addresses that are very likely to be vulnerable, *e.g.*, a list of hosts with port 80 open for a worm targeting at a certain type of web servers. During the hitlist-infection phase, the very first infected host scans the IP addresses in the hitlist, and whenever it infects one, it gives away half of the remaining hitlist to the newly infected host so that together they can infect all hosts in the original hitlist quicker. This process repeats, and as a result, if v out of the S addresses in the hitlist turn out to be vulnerable hosts, all those hosts will get infected in $O(\frac{S}{vr}\log_2 v)$ time, where r is the scanning rate. Even for a modestly big hitlist, this time is miniscule compared to the time it will take to infect the rest of the vulnerable hosts outside the hitlist. To illustrate with an example, suppose there are about 1M vulnerable hosts in IPv4 and a worm starts with a hitlist of $S = 10$K hosts, with approximately $v = 5$K of them actually being vulnerable. If the scanning rate r is 1000 addresses/sec, then the time taken to infect the initial 5K hosts in the hitlist will be approximately 0.025 second, which can arguably be ignored compared to the time the worm will take to infect the rest of the vulnerable hosts in the Internet. Thus, to keep the model simple, if the hitlist contains v vulnerable hosts, we assume that all v of them are infected at time t=0.

4.2 Scanzone and Classification of Vulnerable Hosts

We introduce the concept of *scanzone* and classify vulnerable hosts into different categories, which lays the foundation for our analysis in the next section.

4.2.1 Terminology and Notations

We classify infected hosts into two categories: (1) *active infected hosts*, which are actively scanning for vulnerable hosts, and (2) *retired infected hosts*, which have stopped

scanning. When the context makes it clear, we omit "infected" from the above terms.

Other terms are defined as follows:

- **Jump**: When an infected host chooses a random location on the permutation ring to perform its sequential scan along the ring, we say that the host *jumps* (to that location).
- **Old infection**: When an active host hits a vulnerable host h that was infected previously, we denote the event (as well as host h) as an *old infection*.
- **New infection**: When an active host hits a vulnerable host h that was *not* previously infected, we denote the event (as well as host h) as a *new infection*.
- ***k*-Jump worm**: A permutation-scanning worm is called a k-jump worm if an active host, upon hitting an old infection, jumps to a new location on the permutation ring to resume scanning, but it will retire when hitting its $(k+1)^{th}$ old infection. When a vulnerable host not in the hitlist becomes a new infection, it jumps to a random location on the ring to begin its scan. Subsequently this host can make k other jumps after hitting old infections on the ring. For a vulnerable host in the hitlist, it begins scanning from its own location and then it can make k jumps. To summarize, irrespective of whether it was in the hitlist or not, a host in a k-jump worm is allowed to jump for its first k old infections; but when it hits its $(k+1)^{th}$ old infection, it retires.
- **0-Jump worm**: A permutation-scanning worm is called a 0-jump worm if an active host retires upon hitting its very first old infection. It is a special case of k-jump worm with k=0. A vulnerable host not in the hitlist can make one jump when it becomes a new infection itself, but subsequently when it hits an old infection, it will retire immediately.

4.2.2 Scanzone of an Active Infected Host

As an active infected host h scans the addresses along the permutation ring, it leaves behind a contiguous segment of scanned addresses. This contiguous segment, called the *scanzone* of host h, contains the addresses that h has scanned since its last jump or time 0 if h has not jumped yet; it may contain more addresses if scanzone merge happens, which will be discussed shortly. The scanzones of all active hosts cover all addresses scanned so far. The address of each infected host belongs to a scanzone because it is a scanned address. The *front end* of a scanzone is the address that h is currently scanning; the *back end* refers to the address at the other end of the scanzone, which is the first address that

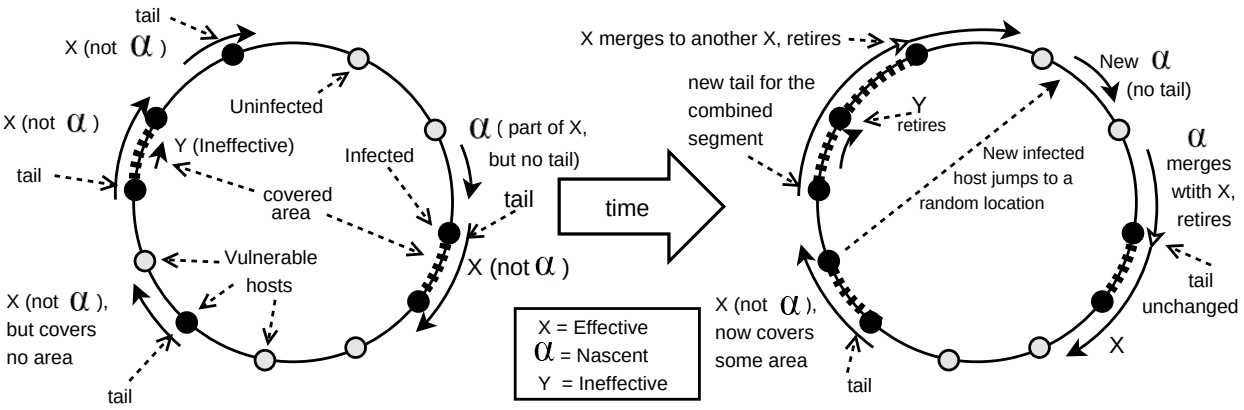

Figure 4-1. Scanzones for a 0-jump worm over time. Scanzones of active hosts are depicted as arcs on the permutation ring. Uninfected and infected vulnerable hosts are depicted as white and dark dots on the permutation ring, respectively.

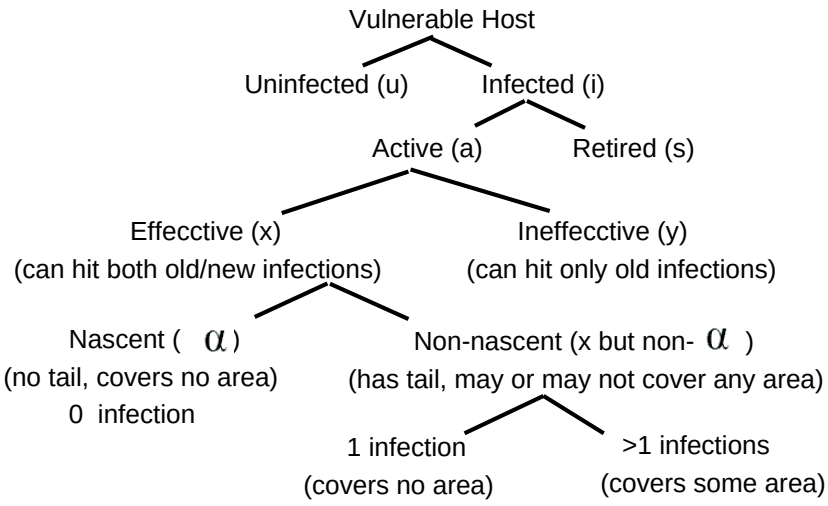

Figure 4-2. Classification of vulnerable hosts for a permutation-scanning worm

h scans after its last jump. Evidently all vulnerable hosts in a scanzone must have been infected. Among all infected hosts in a scanzone, the one that is closest to the back end is called the ***tail*** of the scanzone, and the one that is closest to the front end is called the *head* of the scanzone. The portion of a scanzone between the tail and the head is referred to as the ***covered area*** (portrayed as ▪▪▪▪▪ in Fig. 4-1) of the scanzone. A scanzone may not have a tail (or head) if the active infected host has not hit any vulnerable host

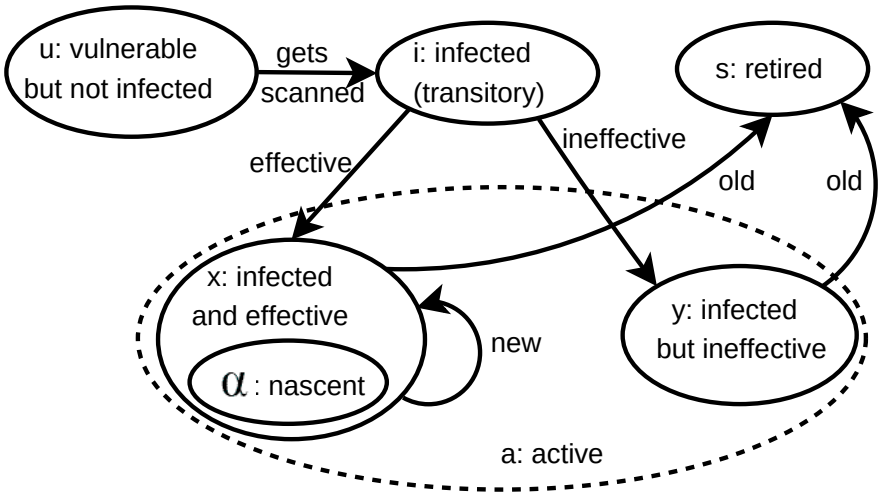

Figure 4-3. Class transition diagram of a 0-jump worm. Here, "new" or "old" indicates the event of a *new* or *old* infection. Similarly, "ineffective" or "effective" indicates whether the newly infected host, after the random jump, lands inside a covered area or not.

since its last jump, and it may not have any covered area if it does not have at least two infected hosts in it.

As h scans more and more addresses, the front end advances to expand the scanzone. But when h hits an old infection h_{old} (which must belong to the scanzone of some active infected host h_1), h surrenders its scanzone by merging it to h_1's scanzone. Then h jumps to a random location to create its new scanzone afresh, or retires if h_{old} is the $(k+1)^{th}$ old infection that it hits. Therefore, the back end of a scanzone may also change if the front end of another scanzone catches up its tail and causes a merge. Merges create larger scanzones. Eventually, all scanzones will be merged into one when all active hosts retire. Only active hosts have scanzones (uninfected or retired hosts do not). We stress that an infected host does not need to know its scanzone; it is an abstract concept used in our mathematical modeling only. The scanzones are shown as arcs on the permutation ring in Fig. 4-1, which also illustrates other concepts to be defined in this section.

4.2.3 Classification of Vulnerable Hosts

In our model, we define classes u, i, a, s, x, y, α for vulnerable hosts that are uninfected, infected, active, retired, effective, ineffective, and nascent, respectively, and

we deliberately make the above class notations the same as the corresponding variables in our later propagation model for the *sizes* of these classes. Fig. 4-2 shows the containment relationship among different classes.

While other classes are self-explaining, we focus on classifying the active hosts, class a, into subcategories, class x and class y, based on whether an active node's scanning is effective or not, i.e., whether it has the potential to generate new infection *before* hitting an old one (note that since the size of the ring is finite, every active host will eventually hit an old infection).

- **Ineffective hosts (class y)**: An active infected host is considered *ineffective* if it is impossible for the host to generate any new infection in future before hitting an old infection. An active host that jumps into a covered area to begin its scanning is evidently ineffective since its first hit will always be an old infection.
- **Effective hosts (class x)**: An active infected host is considered effective if it can *potentially* generate a new infection in future *before* it hits an old one. When an infected host jumps to a point outside of any covered area and starts scanning from that point on, it can potentially generate new infections, and thus called *effective*. The effective hosts are branded as class x. This class is further subdivided as follows:
  - **Nascent hosts (class α):** The effective hosts that are yet to infect any vulnerable host in their *current* scanzones (which, obviously, have no head or tails) are termed as *nascent* (class α). An active host becomes nascent after it takes a jump and lands outside any covered area. Note that after the jump the host starts with a fresh scanzone.
  - **Non-nascent hosts:** Once a nascent host hits a new infection, it becomes a non-nascent effective host; and the host it just infected becomes the tail of its scanzone. Also, each of the initially infected hosts starts as a non-nascent effective host because its scanzone has a tail from the very beginning (the active host itself).

We observe that every infected host in the address space belongs to the scanzone of a non-nascent effective host. This is true at the beginning as each of the initially infected hosts belongs to its own scanzone. When a non-α effective host h_1 infects another host h_{new}, the address h_{new} becomes part of h_1's scanzone. When h_1 retires by hitting h_{old} (tail of a non-α effective host h_2's scanzone), h_1's scanzone merges with h_2's scanzone, and the infections made in h_1's scanzone now become part of h_2's scanzone. Continuing this way,

every infected host remains part of the scanzone of a non-nascent effective host until the last active host retires. It should be noted that the scanzones of nascent or ineffective hosts do not contain any infected hosts.

Fig. 4-3 gives the class transition diagram for a 0-jump worm. A vulnerable host becomes infected when it is scanned by another infected host. When it jumps, it may be either effective or ineffective, depending on whether it jumps into a covered area or not. An effective host begins as a nascent one and becomes non-nascent once it infects another host. An active host retires upon hitting an old infection. Fig. 4-1 also provides illustration for transitions among different classes.

4.3 Modeling the Propagation of 0-Jump Worms

In this section, we derive a series of differential equations that together form the propagation model of 0-jump worms. We extend it for k-jump worms in the next section.

4.3.1 Important Quantities in Modeling

The propagation model of a worm reflects the fractions of vulnerable hosts that are infected, active and retired over time. A scan message that does not hit any vulnerable host does not change these numbers. Thus, modeling should only be based on the event of a scan message hitting a vulnerable host. When that event happens, all aforesaid numbers change. We derive the model by analyzing the precise amounts by which they change. To model a 0-jump worm mathematically, we have to compute the following quantities:

Q1: Between time t and $t+dt$ (for an infinitesimally small dt), how many vulnerable hosts will an active host hit with its scan messages?

Q2: When an effective host hits a vulnerable host h, what is the probability that h is an old infection, and what is the probability that h is a new infection? Note that an ineffective host, by definition, never hits a new infection.

Q3: After a newly infected host jumps, what is the probability for it to be ineffective, and what is the probability for it to be effective?

Table 4-1. Basic notations used for propagation modeling.

N	Size of the address space
V	Total number of vulnerable hosts
v	Number of initially infected hosts (same as the number of vulnerable hosts in a hitlist)
ϕ	$\frac{v}{V}$, the fraction of vulnerable hosts that are initially infected
r	Rate at which an infected host scans the address space
$u(t)$	Fraction of vulnerable hosts that are uninfected at time t
$i(t)$	Fraction of vulnerable hosts that are infected at time t
$a(t)$	Fraction of vulnerable hosts that are actively scanning at time t
$x(t)$	Fraction of vulnerable hosts that are actively scanning at time t and have a non-zero probability of finding new infections
$y(t)$	Fraction of vulnerable hosts that are actively scanning at time t and have a zero probability of finding new infections
$\alpha(t)$	Fraction of vulnerable hosts that are actively scanning at time t with a a non-zero probability of finding new infections but are yet to hit any infection
$s(t)$	Fraction of vulnerable hosts that have retired from scanning

4.3.2 Determining the Quantities Using Probabilistic Approach

Before we delve into deriving the aforesaid quantities mathematically, first we summarize the notations that we would use in this chapter (as well as in chapter 5 in Table 4-1 for quick reference.

As evident from Table 4-1, we use $u(t)$, $i(t)$, $a(t)$, $s(t)$, $x(t)$, $y(t)$ and $\alpha(t)$ to denote the fractions of vulnerable host population that are uninfected, infected, active, retired, effective, ineffective and nascent at time t, respectively. From Fig. 4-2, it is easy to see that $u(t) + i(t) = 1$, $i(t) = a(t) + s(t)$, and $a(t) = x(t) + y(t)$.

Answer for Q1: Let $\boldsymbol{f_{hit}}$ be the number of vulnerable hosts that an active host is expected to hit during a period of dt after time t. Since vulnerable hosts are uniformly distributed on the permuted address space due to randomization of the permutation process, every address on the permutation ring has a probability of $\frac{V}{N}$ to be a vulnerable host. An active host scans $r \times dt$ addresses during dt period. Hence, we have $f_{hit} = r \times dt \times \frac{V}{N}$. Note that the vulnerable hosts that are hit may include both new and old infections.

Answer for Q2: When an effective host hits a vulnerable host, let $\boldsymbol{f_{new}(t)}$ ($\boldsymbol{f_{old}(t)}$) denote the probability for the vulnerable host to be a new (old) infection. We observe that an effective host can hit only two types of vulnerable hosts: 1) those that are uninfected, and 2) infected ones that are the *tails* of scanzones for non-α effective hosts. Recall that scanzones of nascent or ineffective hosts do not have tails. At time t, there are $V(1-i(t))$ uninfected vulnerable hosts (possible new infections) and $V(x(t)-\alpha(t))$ tails (possible old infections). Hence, the chance for hitting a new infection is $f_{new}(t) = \frac{V(1-i(t))}{V(1-i(t))+V(x(t)-\alpha(t))} = \frac{(1-i(t))}{(1-i(t))+(x(t)-\alpha(t))}$, and $f_{old}(t) = 1 - f_{new} = \frac{(x(t)-\alpha(t))}{(1-i(t))+(x(t)-\alpha(t))}$.

Answer for Q3: After a newly infected host jumps to a random location to begin its scanning, let $\boldsymbol{f_{ineff}(t)}$ ($\boldsymbol{f_{eff}(t)}$) be the probability for the host to be ineffective (effective). Since a host becomes ineffective when it jumps into a covered area, $f_{ineff}(t)$ must be equal to the fraction of the permutation ring that all covered areas together represent. Because vulnerable hosts are distributed randomly on the ring, it must also be equal to the fraction of vulnerable hosts that are located in the covered areas, excluding tails because, if we use the number of vulnerable hosts in a covered area to represent its length (in a statistical sense), we cannot count both head and tail that delimits the two ends of the area (a scanzone with a single infection can be thought of having a covered area of length 0). All infected hosts, $Vi(t)$ of them, are located in the covered areas, and there are $V(x(t)-\alpha(t))$ tails because every non-nascent effective host has a scanzone with a tail by definition. Therefore, $f_{ineff}(t) = \frac{Vi(t)-V(x(t)-\alpha(t))}{V}$, and $f_{eff}(t) = 1 - f_{ineff}(t)$.

4.3.3 Propagation Model

We now derive how $i(t)$, $a(t)$, $s(t)$, $x(t)$, $y(t)$ and $\alpha(t)$ change over time t. Below we compute the amounts, $di(t)$, $da(t)$, $ds(t)$, $dx(t)$, $dy(t)$ and $d\alpha(t)$, by which they change respectively over an infinitesimally small dt after time t. This will give us a set of differential equations that together characterize the propagation of 0-jump worms.

- $\boldsymbol{di(t)}$: This, when multiplied by V, represents the total number of new infections generated during dt. Only effective (class x) hosts can hit new infections. The

number of vulnerable hosts hit by effective hosts over dt is $Vx(t)\ f_{hit}$, and each of them has a probability of $f_{new}(t)$ to be a new infection. Hence $di(t) = x(t)\ f_{hit}\ f_{new}(t)$.

- $\boldsymbol{dx(t)}$: Each of the $x(t)f_{hit}f_{new}(t)V$ new infections has a probability of $f_{eff}(t)$ to be effective. This adds $x(t)f_{hit}f_{new}(t)Vf_{eff}(t)$ new effective hosts after dt. On the other hand, effective hosts together hit $x(t)\ f_{hit}\ f_{old}(t)V$ old infections during dt, each causing an effective host (that hits the old infection) to retire. Combining the above two numbers and representing the gross change in fraction, we have $dx(t) = x(t)\ f_{hit}\ f_{new}(t)\ f_{eff}(t) - x(t)\ f_{hit}\ f_{old}(t)$.

- $\boldsymbol{d\alpha(t)}$: Each nascent host (which is effective by definition) is no longer nascent once it hits any vulnerable host. Each of its $r \times dt$ scan messages has a $\frac{V}{N}$ probability of hitting a vulnerable host. Hence, the probability for a nascent host to become non-nascent over dt is $r \times dt \times \frac{V}{N} = f_{hit}$ because, as dt approaches to zero, the joint probabilities for two or more hits are negligible. This reduces the number of nascent hosts by $\alpha(t)Vf_{hit}$. On the other hand, since all new effective hosts created during dt start as nascent, we have $x(t)V\ f_{hit}\ f_{new}(t)\ f_{eff}(t)$ new nascent hosts. Combining these two numbers and representing the gross change in fraction, we have $d\alpha(t) = x(t)\ f_{hit}\ f_{new}(t)\ f_{eff}(t) - \alpha(t)\ f_{hit}$.

- $\boldsymbol{dy(t)}$: Recall that whenever a host jumps into a covered area, it becomes ineffective. For a 0-jump worm, only the newly infected hosts make a jump and thus only they may increase $y(t)$. There are $x(t)Vf_{hit}f_{new}(t)$ new infections, and each has a probability of $f_{ineff}(t)$ to become ineffective. On the other hand, when an existing ineffective host hits a vulnerable host, it retires since ineffective hosts can hit old infections only. Combining these two factors and representing the gross change in fraction, we have $dy(t) = x(t)f_{hit}f_{new}(t)f_{ineff}(t) - y(t)f_{hit}$.

- $\boldsymbol{ds(t)}$: Whenever an effective host hits an old infection, or an ineffective host hits *any* vulnerable host (which must be an old infection), it retires. Within time dt, there are $x(t)Vf_{hit}f_{old}(t) + y(t)Vf_{hit}$ newly retired hosts, and thus $ds(t) = x(t)f_{hit}f_{old}(t) + y(t)f_{hit}$.

From the above analysis, we have

$$
\begin{aligned}
f_{hit} &= r \times dt \times \frac{V}{N} && (4\text{–}1)\\
f_{old}(t) &= \frac{x(t) - \alpha(t)}{1 - i(t) + x(t) - \alpha(t)} && (4\text{–}2)\\
f_{new}(t) &= \frac{1 - i(t)}{1 - i(t) + x(t) - \alpha(t)} = 1 - f_{old}(t) && (4\text{–}3)\\
f_{ineff}(t) &= i(t) - (x(t) - \alpha(t)) && (4\text{–}4)\\
f_{eff}(t) &= 1 - i(t) + x(t) - \alpha(t) = 1 - f_{ineff}(t) && (4\text{–}5)\\
di(t) &= x(t)\ f_{hit}\ f_{new}(t) && (4\text{–}6)\\
dx(t) &= x(t)\ f_{hit}\ f_{new}(t)\ f_{eff}(t) - x(t)\ f_{hit}\ f_{old}(t) && (4\text{–}7)\\
d\alpha(t) &= x(t)\ f_{hit}\ f_{new}(t)\ f_{eff}(t) - \alpha(t)\ f_{hit} && (4\text{–}8)\\
dy(t) &= x(t)\ f_{hit}\ f_{new}(t)\ f_{ineff}(t) - y(t)\ f_{hit} && (4\text{–}9)\\
ds(t) &= x(t)\ f_{hit}\ f_{old}(t) + y(t) f_{hit} && (4\text{–}10)\\
da(t) &= dx(t) + dy(t). && (4\text{–}11)
\end{aligned}
$$

The boundary condition to these set of equations are: $i(0) = a(0) = x(0) = \frac{v}{V}$, and $\alpha(0) = s(0) = y(0) = 0$, where v is the number of vulnerable hosts in the hitlist.

4.3.4 Verification of Our Model

We developed a packet-level simulator for permutation worms whose propagation strategies are described in Section 4.1.2. The simulator is implemented in C++ with proper encapsulation, i.e., a host object inside the simulator is not aware of the large picture of the network, and instead it can only see its own private variables, including its IP address, the state of its local random-number generator, the last address scanned, and the response to a scan message, i.e., new infection or not. The controller object of the worm simulator performs the initial infection, and does the high-level counting of infected, active and retired hosts at the end of each time tick. Each vulnerable-host object uses the same encryption/decryption key but has a different seed for the random number generator used for calculating the random location from which the host, after being infected, will

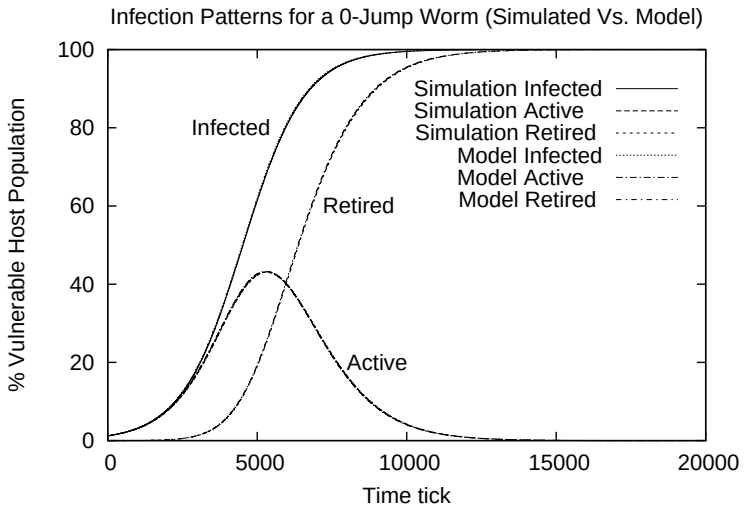

Figure 4-4. Propagation curves for a 0-jump worm (model vs. simulated). The "Model" curves show the percentages of vulnerable hosts that are *infected*, *active*, and *retired* over time, respectively. These curves for $i(t)$, $a(t)$ and $s(t)$ are numerically computed from the analytical model in (4–1)-(4–11). The "Simulation" curves are plotted using the averaged data collected from the packet-level simulator; the 99% confidence intervals are also plotted for selected data points. As expected, the curves from the model and the curves from the simulator completely overlap, which verifies the correctness of the model.

begin its scanning. The simulation stops when all infected hosts retire. Fig. 4-4 compares the propagation curves produced by the simulator with those generated by the analytical model for a 0-jump worm; the two sets of curves are nearly indistinguishable.

The simulation parameters are given as follows: The size of the vulnerable population is $V = 2^{13}$. The hitlist contains $v = 100$ vulnerable hosts. The size of the address space is $N = 2^{23}$; it will take prohibitively long time if N is chosen to be 2^{32}. To produce propagation curves in any of the figures in the dissertation, we simulate worm propagation for 1000 times under different random seeds, and then take the average. We normalize the time tick to be $\frac{1}{r}$. Namely, an infected host sends one scan message per tick. This allows the same propagation curves to be used for characterizing worm propagation under any scanning rate. Hosts with variable scanning rates will be investigated in in Section 4.7.

Worm propagation happens among end hosts. It is not necessary to explicitly simulate the network topology. Because we are particularly interested in stealthy worms that scan at a low rate (e.g., one scan message every few seconds), we assume that the time tick — which is the inverse of scan rate — is larger than the Internet end-to-end delay (typically in tens or hundreds of milliseconds). In this case, infection will be completed within the current time tick, and the impact of the propagation delay of scan messages will be very small on the infection curve, which describes the percentage of vulnerable hosts that are infected over time. As we will further discuss in Section 4.7, even when the Internet end-to-end delay is larger than the time tick, its impact on worm propagation is still small and quantifiable. We will also address other practical considerations, such as host patch and crash, network congestion and bandwidth variability, in Section 4.7.

4.4 Extending the Model to k-Jump Worms

In this section, we demonstrate the flexibility of our analytical model by extending it for the k-jump worms. Modeling the propagation of k-jump worms is important as it will lead to a better understanding of the Warhol worm, which can infect the whole of Internet in a matter of minutes [63]. Warhol worms and Permutation-scanning k-jump worms with large hitlists share similar propagation characteristics.

4.4.1 Further Classification of Active Hosts for k-Jump Worms

For a 0-jump worm, at any time t, none of the $a(t)$ active hosts has hit any old infection. However, for a k-jump worm, any active host (class x, α or y) could have hit anywhere between 0 to k old infections. While the terms $x(t)$, $\alpha(t)$ and $y(t)$ continue to denote the fractions of all vulnerable hosts that are effective (class x), nascent (class α) and ineffective (class y) respectively at time t, for a k-jump worm, each of those classes is further subdivided into k+1 subclasses depending on how many old infections they have already hit (between 0 and k). For example, class x is subdivided into classes x_0, x_1, x_2 ... x_{k-1}, x_k such that $x(t) = \sum_{j=0}^{k} x_j(t)$, and similar notations are used for class α and y.

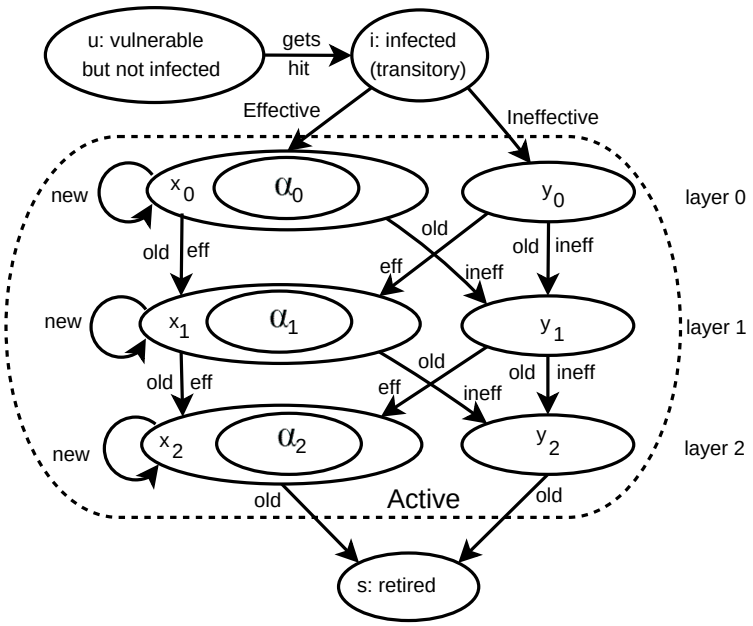

Figure 4-5. State diagram of a k-jump worm with k=2. In the diagram, we assign each active host a *layer number*, which indicates the number of old infections hit by the host. Once the host hits its k+1th old infections, it retires immediately.

For example, the total number of nascent hosts that have hit 2 old infections till time t are denoted by $\alpha_2(t)$.

Fig. 4-5 shows the state diagram that depicts how an active host moves from one subclass to another until it is retired. Each active host is assigned a *layer number*, which indicates the number of old infections hit by the host. An active host having already hit j old infections are referred to as j-layer hosts. When a j-layer host hits another old infection, it moves to layer $j+1$ or to the retired class if $j = k$.

We observe that the quantities, $f_{old}(t)$, $f_{new}(t)$, $f_{eff}(t)$ and $f_{ineff}(t)$ (defined in Section 4.3.2), stay the same for both 0-jump worms and k-jump worms. The analysis that produces the formulas for their calculation can be applied to both 0-jump worms and k-jump worms.

4.4.2 Interaction among Scanning Hosts at Different Layers

The state transitions in Fig. 4-5 between subclasses at different layers are explained below.

- An active infected host never changes its layer by hitting a *new* infection. This is because the layer of a host indicates how many *old* infections the active host has hit till that time, and hitting a new infection does not change that. However, when it hits an old infection, it takes a jump, moves to the next layer and becomes either nascent or ineffective depending on whether it jumps into a *covered area* or not. However, if it was already at the k-layer, then it retires after hitting its $(k+1)^{th}$ old infection.

- Active hosts at *any* layer can hit a new infection. Therefore, when calculating change in $x_0(t)$, $\alpha_0(t)$ and $y_0(t)$, we must consider the new infections caused by effective hosts at all $k+1$ layers.

- The number of active hosts at any layer, except for layer 0, will be changed only by the activity of the hosts at the same or previous layer. The number of hosts at a layer increases when hosts in the previous layer hit old infections and consequently move to this layer. Similarly, it decreases when hosts in this layer hit old infections and move to the next layer. Therefore, the derivative of the number of j-layer hosts, for $1 < j \leq k$, depends only on the numbers of hosts in layer j and layer $j-1$.

4.4.3 Propagation Model for k-Jump Worms

Below we present the equations that model the propagation of k-jump worms. For the purpose of brevity, all symbols used in the formulas are function of time t, except for f_{hit}, V and N, which are independent of time. For example, f_{new} denotes $f_{new}(t)$, $d\alpha_j$ denotes $d\alpha_j(t)$, and so on. We omit the equations for f_{hit}, $f_{old}(t)$, $f_{new}(t)$, $f_{eff}(t)$ and $f_{ineff}(t)$ since they are the same as (4–1)-(4–5). The differential equations for k-jump worms are

$$dx_j(t) = \begin{cases} \text{if } j = 0, & x(t) f_{hit} f_{new}(t) f_{eff}(t) - x_j(t) f_{hit} f_{old}(t); \\ \text{if } 0 < j \leq k, & x_{j-1}(t) f_{hit} f_{old}(t) f_{eff}(t) - x_j(t) f_{hit} f_{old}(t) \\ & + y_{j-1}(t)\ f_{hit}\ f_{eff}(t) \end{cases} \tag{4–12}$$

$$d\alpha_j(t) = \begin{cases} \text{if } j = 0, & x(t) f_{hit}\ f_{new}(t)\ f_{eff}(t) - \alpha_j(t)\ f_{hit}\ ; \\ \text{if } 0 < j \leq k, & x_{j-1}(t) f_{hit}\ f_{old}(t)\ f_{eff}(t) - \alpha_j(t)\ f_{hit} \\ & + y_{j-1}(t)\ f_{hit}\ f_{eff}(t) \end{cases} \tag{4–13}$$

$$dy_j(t) = \begin{cases} \text{if } j = 0, & x(t) f_{hit}\ f_{new}(t)\ f_{ineff}(t) - y_j(t)\ f_{hit}\ ; \\ \text{if } 0 < j \leq k, & x_{j-1}(t) f_{hit} f_{old}(t) f_{ineff}(t) - y_j(t) f_{hit} \\ & + y_{j-1}(t)\ f_{hit}\ f_{ineff}(t) \end{cases} \tag{4–14}$$

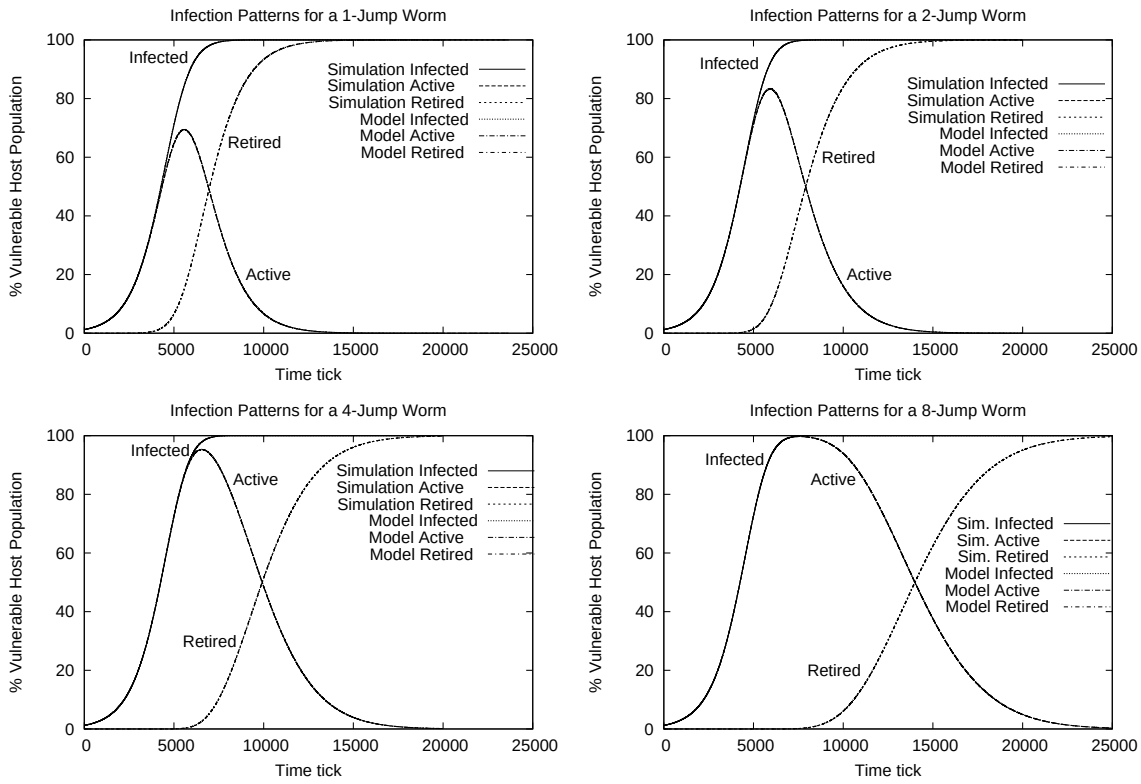

Figure 4-6. Propagation curves for k-jump worms (model vs. simulated). The "Model" curves show the percentages of vulnerable hosts that are infected, active, and retired over time, respectively. These curves for $i(t)$, $a(t)$ and $s(t)$ are numerically computed from the analytical model in (4–12)-(4–20). The "Simulation" curves are plotted using the averaged data collected from the packet-level simulator. As expected, for $k = 1$, 2, 4, and 8, the curves from the model and the curves from the simulator completely overlap, which verifies the correctness of our model for k-jump worms.

$$dx(t) = \sum_{j=0}^{k} dx_j(t) \tag{4–15}$$

$$dy(t) = \sum_{j=0}^{k} dy_j(t) \tag{4–16}$$

$$d\alpha(t) = \sum_{j=0}^{k} d\alpha_j(t) \tag{4–17}$$

$$di(t) = x(t)\ f_{hit}\ f_{new}(t) \tag{4–18}$$

$$da(t) = dx(t) + dy(t) \tag{4–19}$$

$$ds(t) = x_k(t) f_{hit}\ f_{old}(t) + y_k(t) f_{hit}. \tag{4–20}$$

The boundary conditions at time $t = 0$ are $i(0) = a(0) = x(0) = x_0(0) = \frac{v}{V}$. All the other quantities ($s(0)$, $x_1(0) \ldots x_k(0)$, $\alpha(0)$, $\alpha_0(0) \ldots \alpha_k(0)$, $y(0)$, $y_0(0) \ldots y_k(0)$) are zeroes.

4.4.4 Verification of the Correctness of the Model

For different values of k, we compare the result numerically computed from the model with the result collected from the packet-level simulator in Fig. 4-6, using the same experimental setup as described in Section 4.3.4. In all cases, the model and the simulation produce the same propagation curves.

4.5 Closed-Form Solution for the 0-Jump Worm

In this section, we transform the set of equations in (4–1)-(4–11) to three simple differential equations that can be further integrated into the closed-form formulas for the numbers of infected, active and retired hosts over time.

First we establish a functional relation between $i(t)$ and $x(t)$. Recall that $i(t)$ is the fraction of the vulnerable host population that is infected at time t and $x(t)$ is the fraction of the vulnerable host population that is actively scanning and can potentially generate new infections – more precisely, these so-called effective hosts are currently scanning addresses outside of any covered area. By definition, $i(t) = x(t) + y(t) + s(t)$. The infected hosts include effective hosts, ineffective hosts and retired hosts.

We define a *current position* for each infected host. For an effective or ineffective host, its current position is the address it is scanning. For a retired host, its current position is the address it has scanned last before retirement. Interestingly, the current positions of all infected hosts are distributed along the permutation ring uniformly at random. That is because, right after infection, a host jumps to a location that is independently and randomly selected. As long as all infected hosts begin their scanning at independently random locations, their current positions will always be uncorrelated and statistically distributed along the ring uniformly at random.

By definition, the current position of an effective host will be outside of any covered area, and the current position of an ineffective or retired host will be in a covered area. Due to the random distribution of the current positions of all infected hosts, the fraction of infected hosts being effective is equal to the *fraction* of the permutation ring that is

outside of the covered areas, *which* is simply $f_{eff}(t)$. From Section 4.3.2, we know that $f_{eff}(t) = 1 - f_{ineff}(t)$ and $f_{ineff}(t)$ equals the fraction of the ring that all covered areas together represent. Summarizing the above analysis, we have

$$x(t) = i(t)\ f_{eff}(t) \tag{4–21}$$

By plugging the above equation into (4–1)-(4–11), it can be easily verified that this equation is consistent with others in the model.

Applying (4–21), (4–1), (4–3) and (4–5) to (4–6), we have the following differential equation.

$$\frac{di(t)}{dt} \quad = \quad \frac{rV}{N} \times i(t) \times \big(1 - i(t)\big) \tag{4–22}$$

Applying (4–7) and (4–9) to (4–11), we have

$$da(t) = x(t) f_{hit} f_{new}(t) - x(t) f_{hit} f_{old}(t) - y(t) f_{hit}$$

Because $y(t) = a(t) - x(t)$ by definition and $f_{old}(t) = 1 - f_{new}(t)$, we have

$$da(t) = 2x(t) f_{hit} f_{new}(t) - a(t) f_{hit}$$

Applying (4–21), (4–1), (4–3) and (4–5), we have

$$\frac{da(t)}{dt} = \frac{rV}{N} \times \Big(2\ i(t) \times \big(1 - i(t)\big) - a(t)\Big) \tag{4–23}$$

Because $s(t) = i(t) - a(t)$ by definition, $\frac{ds(t)}{dt} = \frac{di(t)}{dt} - \frac{da(t)}{dt}$. From (4–22) and (4–23), we have

$$\frac{ds(t)}{dt} = \frac{rV}{N} \times \Big(a(t) - i(t) \times \big(1 - i(t)\big)\Big) \tag{4–24}$$

Let $\phi = \frac{v}{V}$, which is the fraction of the vulnerable host population that is initially infected at time $t = 0$. Integrating (4–22), (4–23) and (4–24), we have the following close-form solution.

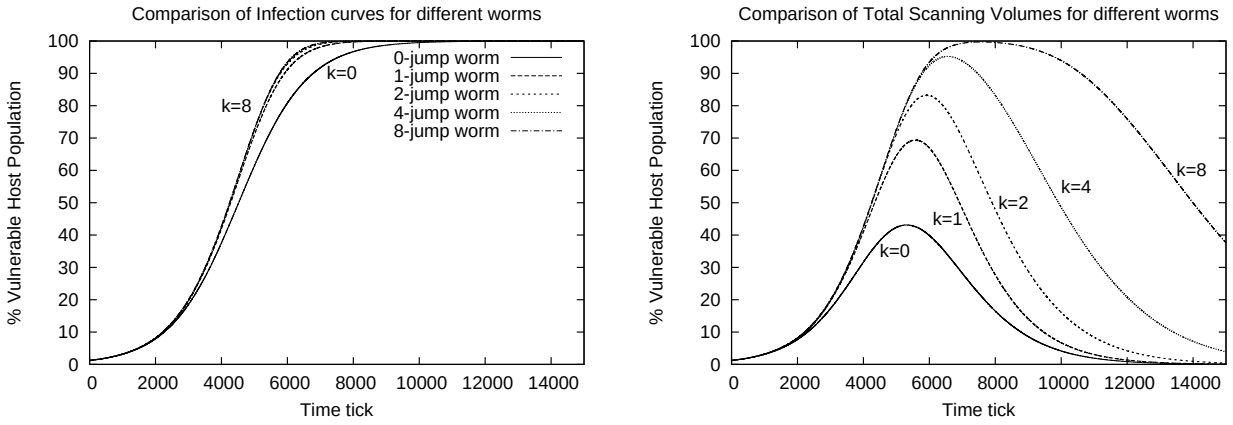

Figure 4-7. Comparison of the infection rates and the total scanning volumes for different k-jump worms. The left plot shows the infection curves, $i(t)$, for a k-jump worm under different k values. The right plot shows the active curves, $a(t)$, for a k-jump worm under different k values. Recall that $a(t)$ is the percentage of vulnerable hosts that are actively scanning at time t. The total amount of scanning traffic, called the *scanning volume*, is defined as the area under the active curve. From the left plot, we see that the infection speed improves when k increases. However, the rate of improvement diminishes quickly when k is greater than 1. On the other hand, from the right plot, the scanning volume increases *significantly* when k increases.

$$i(t) = \frac{\phi e^{\frac{rV}{N}t}}{1-\phi+\phi e^{\frac{rV}{N}t}} \tag{4–25}$$

$$a(t) = \frac{2(1-\phi)}{\phi e^{\frac{rV}{N}t}}\left\{\frac{1-\phi}{1-\phi+\phi e^{\frac{rV}{N}t}} - 1 + \phi + \ln(1-\phi+\phi e^{\frac{rV}{N}t}) + \frac{\phi^2}{2(1-\phi)}\right\} \tag{4–26}$$

$$s(t) = \frac{\phi e^{\frac{rV}{N}t}}{1-\phi+\phi e^{\frac{rV}{N}t}} - \frac{2(1-\phi)}{\phi e^{\frac{rV}{N}t}}\left\{\frac{1-\phi}{1-\phi+\phi e^{\frac{rV}{N}t}} - 1 + \phi + \ln(1-\phi+\phi e^{\frac{rV}{N}t}) + \frac{\phi^2}{2(1-\phi)}\right\} \tag{4–27}$$

4.6 Usage of the Analytical Model

In this section, we first describe the benefits of having an analytical model when comparing with a simulator. We then analyze our model to see what impact each worm/network parameter (such as network size, vulnerable population size, hitlist size and scanning rate) will have on the worm propagation.

4.6.1 Analytical Modeling or Simulation?

Properly simulating the worm propagation on the Internet at the packet level is very difficult due to its sheer scale. Even for a rather simplified version of the Internet, without an analytical model one will need to take the average of multiple runs of a simulator in order to get acceptably reliable propagation curves. And since each run could potentially take a long time for realistic values of N and V, the whole process could take an enormous amount of time. For an imagined attack targeting at the Windows system, it took 16 hours on an Intel Xeon 2.8 GHz processor with 4GB RAM to run a *single* round of a simulation involving around 400M potentially vulnerable windows hosts on IPv4 for one set of worm/network parameters. In order to run the same simulation for IPv6 ($N = 2^{128}$), it is easy to see that the runtime will be astronomical. On the contrary, a *single* run of the numerical simulation based on the analytical model takes just seconds and gives us the provably correct results. Moreover, it can handle extremely large address spaces and vulnerable host populations. For any worm/network parameter change, new propagation curves can be re-computed in little time for comparison.

While arguments can be made for doing a scaled-down simulation and then scaling up the results, such simulations are often not fully accurate and suffer from stochastic fluctuations and other problems [75]. Moreover, such simulations cannot predict with confidence what *precise* effect each worm/network parameter will have on the overall outcome, and for what reason. In comparison, an analytical model can tell *exactly* why and by how much will a parameter affect the outcome.

4.6.2 Impact of the Worm/Network Parameters on a Worm's Propagation

Below we analyze the exact effect of each worm/network parameter on the worm propagation.

- **Effect of Address Space Size (N)**: For either 0-jump worms or k-jump worms, the only term in the model that is directly affected by N is $f_{hit} = r \times dt \times \frac{V}{N}$ in (4–1). Since all the incremental terms (such as $di(t)$, $da(t)$ and $ds(t)$) are direct multiples of f_{hit}, the first derivatives of the propagation curves for infected, active

and retired hosts are inversely proportional to N. The first derivative characterizes the rate of change over time. Therefore, as the size of the address space increases, a worm propagates inverse-proportionally slower. If the address space is doubled, it will take the worm double the amount of time to infect all vulnerable hosts. This gives a reason for transition to IPv6.

- **Effect of Vulnerable Host Population Size (V)**: For either 0-jump worms or k-jump worms, the only term in the model that is affected by V is $f_{hit} = r \times dt \times \frac{V}{N}$. Again because the incremental terms (such as $di(t)$, $da(t)$ and $ds(t)$) are direct multiples of f_{hit}, the first derivatives of the propagation curves for infected, active and retired hosts are proportional to V. As V increases, a worm propagates proportionally faster. If V is doubled, it takes the worm half the amount of time to infect all vulnerable hosts.

- **Effect of Hitlist Size (v)**: The size of the hitlist is controlled by the worm designer. As per our observations from the analytical model, a higher v simply shifts the infection curve, $i(t)$, to the left on the time axis with $i(0) = \frac{v}{V}$. From the infection curve in Fig. 4-4, we see that there is a slow start phase where $i(t)$ increases slowly before it transitions into a rapid growth phase. A larger hitlist will shorten the initial slow start phase and may significantly reduce the overall propagation time.

- **Effect of Scanning Rate (r)**: Again, for either 0-jump worms or k-jump worms, the only term in the model that is affected by r is $f_{hit} = r \times dt \times \frac{V}{N}$. Since the incremental terms (such as $di(t)$, $da(t)$ and $ds(t)$) are direct multiples of f_{hit}, the first derivatives of the propagation curves for infected, active and retired hosts are inversely proportional to r. Thus, if the scanning rate is doubled, the time it takes a worm to infect the vulnerable host population will be halved.

- **Effect of Varying k for a k-jump Worm**: We make an important observation from Fig. 4-7, where the infection speed and the scanning volume of a k-jump worm are presented for different k values. The *scanning volume* is defined as the total scanning traffic generated by all active hosts since time $t = 0$. We see that, by increasing k, the slope of the infection curve in the left plot is somewhat steeper, but for $k > 1$, the incremental gain becomes negligible. On the other hand, with a higher k, the onset of retirement for active hosts happens at a increasingly later time, which means larger scanning volume, as shown in the right plot. We observe that, for $k \geq 8$, almost *all* infected hosts are active at the time when all vulnerable hosts have been infected, producing a big network footprint for worm detection. Therefore, it makes little sense to deploy a k-jump worm with a high value of k.

4.7 Practical Considerations

In this section, we consider our model under real-world considerations, including congestion and bandwidth variability, patching and host crash, as well as delay of scan messages.

4.7.1 Congestion and Bandwidth Variability

If for stealth reason the worm sets a small scanning rate r such as 100 per minute, most infected hosts are likely to have the bandwidth of delivering 100 SYN packets per minute, and our model will be accurate if the deviation caused by Internet delay is negligible. However, if the worm sets its scanning rate r to be 10,000 per second, then the actual scanning rates of infected hosts may vary due to network congestion. We believe a worm that causes network congestion is not a good worm because it loses stealth (unless its sole purpose is to create headlines by service disruption, which is rarely the case nowdays [59]).

Congestion also happens naturally in the network without worm activity due to the bandwidth limitation and the demand on the routers. As long as the Internet is able to deliver the low scanning rate of most infected hosts, our model can predict the propagation behavior of low-rate stealthy worms. However, we realize that whatever be the reason – processing power of infected host, available bandwidth for the user, congestion of the network – the final result is that on the Internet scale, different hosts are in effect scanning at different rates. Therefore, if we can somehow extend our model to accommodate variable scanning rates from different hosts, we are effectively capturing the real network situation arising out of the reasons mentioned above. Since our model can handle only a fixed scanning rate, we posited that by using average scanning rate, our model should be able to still approximate the variable scanning rate scenario. With that goal in mind, we simulated two worms, one having a fixed rate, $r = 5$ per time tick for all infected hosts, and the other having variable rates with the Gaussian distribution and a mean value of 5 per time tick. Fig. 4-8 shows that the infection curves of the two worms

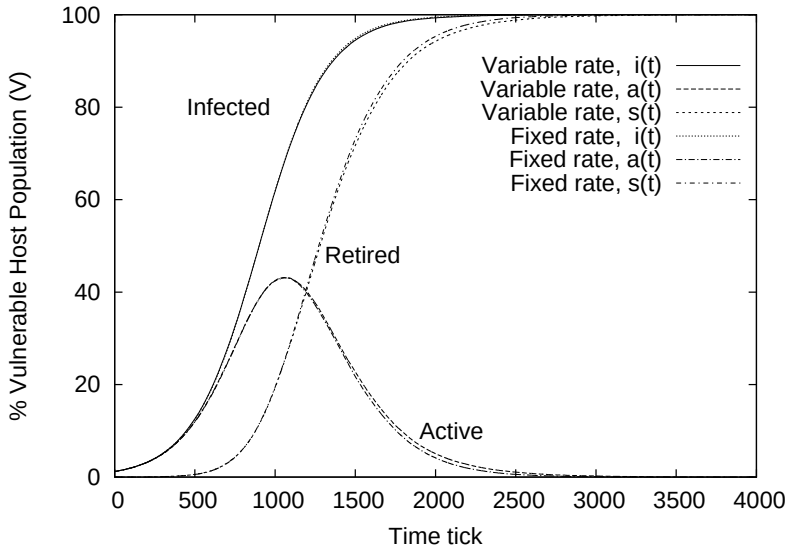

Figure 4-8. Comparison of propagation curves for worms with variable-rate and fixed-rate of scanning. The simulation parameters used are $N = 2^{23}$, $V = 2^{13}$ and $v = 100$. The value of r is a constant 5 scans per time tick for the fixed-rate scanning worm, and the other having variable rates with the Gaussian distribution and a mean value of 5 per time tick.

are very close. Similar results are observed for other variable rate distributions. Therefore, the model is able to approximate the propagation of worms by using average scanning rate. Therefore, we argue that our model is indeed able to approximate the propagation of worms in real-life scenarios by using the average scanning rate.

4.7.2 Patching and Host Crash

Once a vulnerable host is infected and starts scanning, it may be removed from the vulnerable host population due to multiple reasons. First, upon infection the host may simply crash. Second, the host may be patched by the system administrator after some time. Third, due to scanning activity, an infected host may come under suspicion of the network administrator and resultingly taken off the network or quarantined. We show that our model can be extended to handle the removal of vulnerable hosts.

We introduce a few additional terms in our model to account for the removal of hosts. Let p_q denote the probability of a host being removed each time it scans, and $q(t)$ denote the number of vulnerable hosts that are removed from the system by time t. As

hosts are removed, the vulnerable population also changes; we use $V(t)$ for the number of vulnerable hosts at time t. It is evident that $V(t) + q(t) = V(0)$ for all t. However, under this "removal" scheme the meaning of $i(t)$ becomes unclear as some hosts that were infected can later be disinfected. To clear this confusion, we introduce a third new term called $i_{ever}(t)$ to denote the fraction of *original* vulnerable host population ($V(0)$) that were *ever* infected during the whole propagation, while $i(t)$ denotes the fraction of $V(t)$ that are infected. Since $V(t)$ is not a constant, we rather plot $i_{ever}(t)$.

With these new terms, we rewrite the propagation equations of a 0-jump worm by considering host removal:

$$f_{hit}(t) = r \times dt \times \frac{V(t)}{N} \tag{4–28}$$

$$f_{old}(t) = \frac{x(t) - \alpha(t)}{1 - i(t) + x(t) - \alpha(t)} \tag{4–29}$$

$$f_{new}(t) = \frac{1 - i(t)}{1 - i(t) + x(t) - \alpha(t)} = 1 - f_{old}(t) \tag{4–30}$$

$$f_{ineff}(t) = i(t) - (x(t) - \alpha(t)) \tag{4–31}$$

$$f_{eff}(t) = 1 - i(t) + x(t) - \alpha(t) = 1 - f_{ineff}(t) \tag{4–32}$$

$$dx(t) = x(t) f_{hit}(t) f_{new}(t) f_{eff}(t) - x(t) f_{hit}(t) f_{old}(t) - x(t) p_q \tag{4–33}$$

$$d\alpha(t) = x(t) f_{hit}(t) f_{new}(t) f_{eff}(t) - \alpha(t) f_{hit}(t) - \alpha(t) p_q \tag{4–34}$$

$$dy(t) = x(t)\ f_{hit}(t)\ f_{new}(t)\ f_{ineff}(t) - y(t)\ f_{hit}(t) - y(t) p_q \tag{4–35}$$

$$ds(t) = x(t)\ f_{hit}(t)\ f_{old}(t) + y(t) f_{hit}(t) \tag{4–36}$$

$$da(t) = dx(t) + dy(t) \tag{4–37}$$

$$dq(t) = x(t) p_q + y(t) p_q \tag{4–38}$$

$$di(t) = x(t)\ f_{hit}(t)\ f_{new}(t) - dq(t) \tag{4–39}$$

$$di_{ever}(t) = di(t) + dq(t) \tag{4–40}$$

$$dV(t) = -dq(t) \tag{4–41}$$

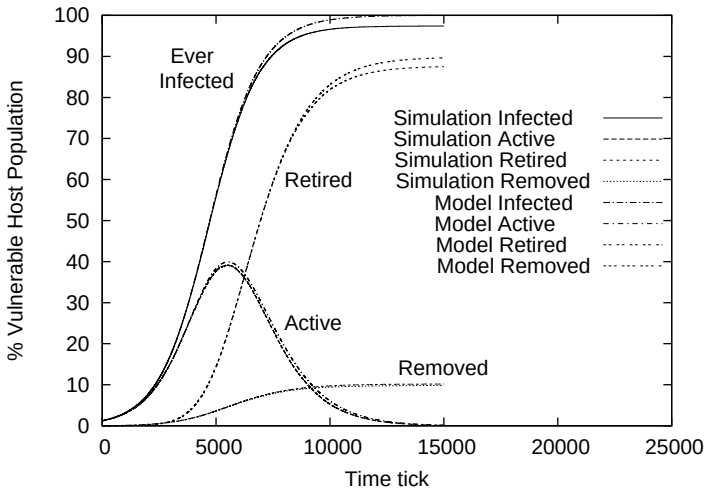

Figure 4-9. Comparison of propagation curves for a 0-Jump worm with removal of hosts (due to patching, quarantining, disconnection, crash, etc.). We use the following parameters: $N = 2^{23}$, $V(0) = 2^{13}$, $r = 1$ per time tick, and $p_q = 0.00005$. The result from the model displays a reasonable match to the results from the simulation for up to 90% infection.

The boundary condition to these set of equations are $i(0) = a(0) = x(0) = \frac{v}{V(0)}$, and $\alpha(0) = s(0) = y(0) = 0$.

4.7.3 Internet Delay

When deriving the propagation model, we implicitly assume that each scan message instantaneously reaches the address being scanned. In reality, the worm will propagate slower due to end-to-end delay of the Internet. Hence, the model in (4–25) gives an upper bound on the worm's propagation speed.

In case of a new infection using TCP, it takes one round trip to exchange SYN (which is the scan message) and SYN/ACK, and then it takes a number of round trips to transmit ACK and attack packets. For example, if the worm code is 3k long and each TCP segment is 512 bytes, then under TCP's slow start it takes three round trips to complete the infection. Internet's round trip delay rarely exceeds one second [16]. Let D be a time period that upper-bounds the delay of most infections. Since worm code is typically short (in order to fit in the call stack without causing the program to crash when buffer-overflow attack is used), D is expected to be no more than several seconds.

The larger the infection delay is, the slower the worm propagates. Hence, if we artificially set the delay of all infections to the upper bound D (ignoring the rare cases whose delay exceeds D), we have a lower bound on the worm propagation speed. It can be shown that this lower bound is simply the infection curve (4–25) shifted to the left by D. Combining both the lower bound and the upper bound, we have the following inequality for the actual value of $i(t)$ after Internet delay is considered. For $t \geq D$,

$$\frac{\phi e^{\frac{rV}{N}(t-D)}}{1-\phi+\phi e^{\frac{rV}{N}(t-D)}} \leq i(t) \leq \frac{\phi e^{\frac{rV}{N}t}}{1-\phi+\phi e^{\frac{rV}{N}t}} \tag{4–42}$$

If a worm wants to stay undetected, it will choose a low scanning rate for better stealthiness (smaller footprint on the Internet) even when that means lower propagation speed and longer propagation time. Many known worms take hours or tens of minutes to infect the Internet. For these worms, a maximum deviation of several seconds by the model from the reality is relatively small with respect to the much longer overall propagation time. Note that our goal here is not to determine the actual value of D. Instead, we argue that the predictive power of our model is relevant in reality when the Internet delay is small comparing with the worm propagation time.

4.8 Contributions

In this work, we have successfully modeled the propagation characteristics of different varieties of permutation-scanning worms. We have first derived the precise propagation model for 0-jump worms, and then extended it for the general k-jump worms. We have also been able to derive the closed-form solution for the 0-jump worms. To verify the correctness of the model, we have compared the results from our model with those obtained from the simulation, and showed that they match perfectly. We have analyzed the model to demonstrate how each worm/network parameter will affect the worm's propagation behavior. Finally, although our analytical model was originally conceived by assuming ideal network conditions, we have showed that it can very well be extended to

real-life scenarios with the consideration of variable host bandwidth, network congestion, Internet delay, host crash and patching.

CHAPTER 5
WORM DESIGN: THE IMPACT OF PSEUDO-RANDOMNESS AND THE OPTIMAL SCANNING STRATEGY

Modeling the propagation of Internet worms has been a very interesting and challenging research area [63, 22, 80, 7, 78, 40]. In the past two decades, worms evolved not only in their actions launched upon their victims but also in their propagation strategies. Knowing the propagation strategy of a worm is important because it allows network administrators and researchers to have a bird's-eye view on the dynamics of the worm's spread across the Internet, which in turn helps them understand how fast any defense protocol must react to contain or eradicate the worm.

Random scanning and its variants [66, 13, 54, 69, 27, 21, 46, 70] are the prevalent method of worm propagation on the Internet. The classical epidemic model has long been established as the tool for characterizing the propagation properties of random-scanning worms [63, 22]. Borrowed from the epidemiology theory, this model assumes that addresses to be scanned are selected independently at random and that every random address has an equal probability of being vulnerable to the worm. However, these assumptions are not true in reality. We observe that real worms use pseudo-random number generators to decide which addresses to be scanned. For example, the following recursive formula, $x_n = 214013 \times x_{n-1} + 2531011 \mod 2^{32}$ was used by Slammer [42] and Witty [35] worms to produce addresses (albeit with bugs in implementation and logic). It is a full-cycle linear congruential generator. With an arbitrary seed x_0, it will produce numbers from a deterministic cycle where each possible number of 32 bits will appear exactly once. Pseudo randomness, however, breaks the underlying assumptions of the epidemic model. The numbers generated are not independent, and our analysis will show that a portion of the worm's scanning activity has *zero* probability of discovering any new infection. Moreover, this portion grows over time. It raises a serious question. Is the epidemic model correct for real worms that perform pseudo-random scanning? To answer this question decisively,

we must incorporate the property of pseudo randomness in the mathematical modeling process, which has not been done in the literature.

Pseudo randomness has another profound impact on worm propagation. Random-scanning worms have a serious weakness: They do not know when the whole address space has been fully scanned (so that they can stop); each address may be scanned again and again. This weakness leads to unnecessarily high volume of scanning traffic, hurting the stealthiness of the worms. Self-stopping technologies have been investigated for the infected hosts to retire from scanning based on a variety of heuristics [39], among which permutation scanning appears to be the best candidate for ensuring that the whole address space is scanned and all vulnerable hosts are infected before the worm stops scanning [63]. In the work, we observe that pseudo randomness (a natural feature of ordinary random number generators) can also be exploited to enable an orderly retirement of unproductive infected hosts. We show that, with a full-cycle random number generator and a termination condition, all existing random-scanning worms can be easily turned into the new *full-cycle worms* that will terminate after covering the entire address space. What makes this work unique is that we derive an *accurate, closed-form*, and *easy-to-interpret mathematical model* that is the first of its kind to characterize the propagation of a self-stopping worm. The detailed propagation model enables us to gain a number of interesting insights into the full-cycle worms (as well as the permutation worms).

First, to our surprise, pseudo randomness does not slow down a worm's propagation even though it causes a portion of the scanning activity to have zero probability of finding new infection. Using a termination condition to retire unproductive infected hosts from scanning, full-cycle worms are stealthier, and remarkably, they achieve stealthiness for free without any sacrifice in their propagation speed. They share *identical* infection curves with ideal random-scanning worms.

Second, the total volume of scan traffic by a full-cycle worm is within a factor of 1.5 from the optimal – which is to scan each address once and only once. (In comparison, a

random-scanning worm will scan each address many times without stopping even after all vulnerable hosts are infected). Also, the maximum number of infected hosts that perform scanning simultaneously (i.e., the peak scanning traffic) is about 43% of the vulnerable population. Interestingly, this number is largely independent of the size of the vulnerable population and the number of the initially infected hosts (which is related to the size of hitlist [63]). Hence, even though the total volume of scan traffic is sub-optimal, the maximum instantaneous scan intensity of a full-cycle worm remains 43% that of a classical random-scanning worm. Even though the duration at maximum scan intensity can be very brief, this result still points out a future direction for defense systems to detect stealthy full-cycle worms — by monitoring the dynamics of instantaneous scan intensity.

Third, based on the model, we conclude that the rate-limit techniques [77, 62, 6] designed against random-scanning worms will perform just as well when they are applied against full-cycle worms. However, the defense techniques targeting at the scanning volume will be less effective.

Fourth, with a simpler design, we show the functional equivalence of full-cycle worms and permutation worms [63], which means our mathematical model for the former and the insights it brings can also be applied to the latter. This is a significant advance over the prior model of permutation worms [40], which consists of a series of inter-dependent differential equations (without any closed-form solution) that are hard to interpret and gain insight from.

We perform simulations to verify our analytical results. We also investigate the applicability of our model under real-world conditions by considering Internet delay and congestion. We show that the model can serve as a reasonably accurate approximation for the propagation of real worms, particularly the stealthy ones that scan at low rates.

The rest of this chapter is organized as follows. Section 5.1 designs full-cycle worms. Section 5.2 describes the criteria we use to characterize the stealthiness of a worm.

Section 5.3 presents the propagation model for full-cycle worms. Section 5.4 analyzes the stealth properties of full-cycle worms. Section 5.6 draws the conclusion.

5.1 Pseudo Randomness and Full-Cycle Worms

We first argue the necessity for incorporating pseudo randomness into the modeling of real worms. We then describe a significant yet easy-to-implement enhancement of these worms, which is enabled by pseudo randomness, allowing the worms to greatly reduce their overall scan traffic and improve stealthiness.

5.1.1 Is the Classical Worm Model Correct?

The epidemic model of infectious diseases has been widely used to characterize worm propagation [63, 22] because most known Internet worms are thought to perform random-scanning (there are exceptions though, like localized subnet scanning, topological scanning etc.). Before we challenge its randomness assumption, we give a brief review below.

Each host infected by a random-scanning worm scans the Internet by repeatedly picking a random address and probing the address to see if the host on that address is vulnerable to a certain attack. If so, it compromises the host by exploiting the vulnerability. As more hosts are infected, their combined rate of scanning increases until all vulnerable hosts are infected.

Using the same notations described in Table 4-1, the classical model for random-scanning worms can be derived as follows. At time t, the number of infected hosts is $i(t)V$, and the number of vulnerable but uninfected hosts is $(1 - i(t))V$. After an infinitesimally small period dt, $i(t)$ changes by $di(t)$. During that time, the number of scan messages made by all infected hosts is

$$n(t) = i(t)Vrdt. \tag{5–1}$$

The probability for one scan message to hit an uninfected vulnerable host is

$$p(t) = (1 - i(t))V/N. \tag{5–2}$$

The number of newly infected hosts, $V di(t)$, is equal to $n(t) \times p(t)$, because the probability of multiple scan messages hitting the same host is negligible when $dt \to 0$. This leads to the following differential equation:

$$\frac{di(t)}{dt} = r\frac{V}{N}i(t)(1 - i(t)) \tag{5–3}$$

Other more sophisticated models take network congestion, dormant state of infected hosts, and localized scanning strategy into consideration [80, 7]. However, as will be discussed in the next section, localized scanning can make a worm more vulnerable to detection, especially when done from outside.

The epidemic model assumes that random addresses selected by infected hosts are independent of each other. It further assumes that each scan message has an equal probability of finding a new infection. However, real worms use pseudo-random number generators to produce addresses for scanning. These addresses are not truly random – they are related through the formula of the generator. It leads to an interesting consequence: Some scan messages will have zero probability of finding new infection, which we will explain below.

We first review the prevalent mechanism behind the popular random number generators [48, 31]. It generates numbers deterministically from a large cycle of numbers arranged in a pseudo-random order (which is designed to pass most of the statistical tests for randomness). The caller supplies an arbitrary seed, which determines the initial position on the cycle to draw the first number. For subsequent numbers, the generator simply walks (say, clockwise) on the cycle. If the cycle contains one and only one occurrence for each number in range $[0..N)$, it is called a *full-cycle* random number generator, where N should be 2^{32} to generate IP addresses. One example of a full-cycle generator is $x_n = 214013 \times x_{n-1} + 2531011 \mod 2^{32}$, used by Slammer and Witty. Another example is given in Appendix A.

All infected hosts scan along the same pseudo-random cycle, starting from different positions due to different seeds. We have the following observation: Once an infected host scans an address that has already been scanned by another infected host, all subsequent addresses that it scans must have been scanned by the latter. This observation is illustrated in Fig. 5-1, where hosts a and b are infected, a begins its scan at address u, and b begins at address v. They scan the addresses (clockwise along the cycle) that are returned by a random number generator. After a reaches v, since each subsequent address on the cycle has already been scanned by b, host a's scan messages will have zero probability of finding new infection, violating the assumption on which the classical worm model is based.

With a broken assumption, is the epidemic model still right? If the cycle of the random number generator is so huge that the scanning segments of different infected hosts do not overlap (e.g., in Fig. 5-1, if the worm activity dies down before host a reaches v), then Eq (5–3) remains a valid model for the worms. However, the impact of pseudo randomness becomes significant in modeling when a full-cycle generator of size 2^{32} is used. As we will discuss next, future worms will have good reasons to adopt such full-cycle generators (as Slammer did, though with faulty implementation).

5.1.2 Will Pseudo Randomness Make Worms More Powerful?

Random-scanning worms have a serious weakness. The infected hosts do not know when they should stop scanning. They may repetitively probe the same address again and again. After most vulnerable hosts are infected, while the rate of new infections slows down considerably, the scanning activity of all infected hosts is however reaching its peak, leaving a large footprint on the Internet and making its presence obvious.

Pseudo randomness not only makes it harder for us to derive the propagation model of Internet worms, but also gives them the potential of being far more efficient than they are today. We show that, using a full-cycle random number generator and a termination condition, all Internet worms that perform random-scanning can be easily modified to

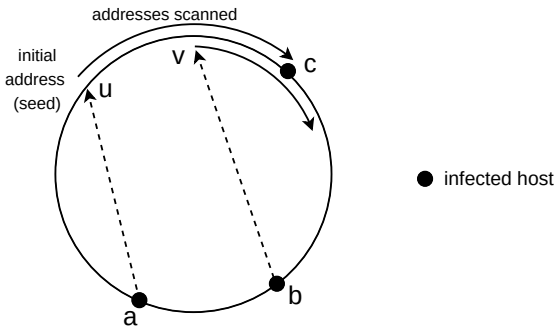

Figure 5-1. Infected hosts scanning during the propagation of a full-cycle worm. Each infected host (a,b), scans a continuous segment of addresses on the cycle. When a's segment reaches b's, the addresses that a will scan have already been covered by b.

solve their weakness. The design is simple: The initially infected hosts that start the worm propagation use their own addresses as the seeds for the full-cycle generator, and each subsequently infected host selects an arbitrary seed. An infected host scans the sequence of addresses produced by the generator. It stops scanning (*retires*) when it reaches a host that has already been infected. After the infected host stops scanning, it is called a *retired host*; before that, it is called an *active host*. These improved worms are called *full-cycle worms*.

The above design is efficient: It allows an infected host to retire when its scanning effort will not find new infection. Refer to Fig. 5-1. After a reaches v, it scans the addresses that have been scanned by b. Even though its effort is no longer productive, a has no way to know until it reaches an infected host c, which will tell a to retire. By retiring at the earliest feasible moment, a avoids sending scan messages to addresses after c. It is easy to see that each infected host will eventually retire because it walks along the cycle and, in the worst case when there is no other infected host, it will come back to itself (and thus know that it should retire).

There are many questions to be answered. How efficient are full-cycle worms exactly? Will they propagate faster or slower than today's random-scanning worms? How hard is it to detect them? More specifically, what is their footprint on the Internet, in terms of the total scan volume and the maximum combined rate of scanning by all active hosts?

Will the existing defense techniques for random-scanning worms remain effective against full-cycle worms? Before we answer these questions, we briefly discuss on two important metrics for measuring worm performance: propagation speed and stealthiness.

5.2 Propagation Speed and Stealthiness

Good worm design must achieve balance between propagation speed and stealthiness. Higher speed and better stealthiness are two conflicting goals. If every infected host scans at its highest possible rate, the worm's propagation speed is maximized. But this makes the worm's presence obvious. For example, the SQLSlammer worm [13] in 2003 caused widespread network congestion across Asia, Europe and the Americas. Its aggressiveness produced headline news, but was not instrumental to its own survival as enormous resources were committed immediately to clean up the worm. The potential harm that a worm can do is decided not by the headlines it generates, but by its longevity. A slow spreading worm that takes a month to silently infect the Internet and stay undetected for a year can do much more harm than a worm that infects the Internet in a day but is wiped out in the next few days. For a malicious worm, generally speaking, stealth is more important than propagation speed. Hence, a worm may artificially configure its scanning rate to a certain low value in order to evade detection.

Propagation speed can be measured based on the infection curve $i(t)$. Stealthiness of a worm can be measured based on the impact of its scanning traffic on the Internet. Three metrics are *gross footprint*, *maximum instantaneous footprint*, and *footprint concentration*, which capture the total amount, the temporal intensity, and the spatial intensity of the scanning traffic, respectively.

- **Gross footprint:** The total volume of scanning traffic over the entire course of worm propagation.
- **Maximum instantaneous footprint:** The maximum combined rate of scanning by all infected hosts at any time instant.
- **Footprint concentration:** Determined by the distribution of scanning traffic (from one or all infected hosts) over the destination subnets.

The footprint concentration is high when the scan traffic from an infected host is mostly directed to a single subnet (i.e., localized scanning), which makes the host susceptible to be identified by address-scan detectors or other IDSs that look for worms doing subnet scanning. The concentration is low when the scan traffic is distributed uniformly at random over the entire address space, which is what the random-scanning worms do. But random-scanning worms are not stealthy because of its large gross footprint and maximum instantaneous footprint. In the next section, we will show why full-cycle worms can do better.

5.3 Propagation Model of Full-Cycle Worms

In this section, we derive the closed-form solution for the infection curve $i(t)$ of a full-cycle worm, which is the fraction of vulnerable hosts that are infected at time t. In the next section, we will derive the closed-form solution for other worm propagation parameters.

5.3.1 Modeling

We show that the infection curve $i(t)$ of a full-cycle worm is identical to that of a random-scanning worm. Let v be the initial number of infected hosts at time $t = 0$ before scanning begins; a hitlist [63] may be used to increase the value of v. These infected hosts are randomly located in the full cycle, where the addresses in $[0..2^{32} - 1]$ are arranged in a pseudo-random order. Assume they begin from their own locations (using their locations as seeds) and scan along the full cycle. There are $(V-v)$ vulnerable hosts randomly placed in the full cycle. Other than the v initially infected ones, the probability for an arbitrary address to be vulnerable is

$$p_0 = \frac{V - v}{N - v} \tag{5–4}$$

It is true that the vulnerable hosts are not distributed uniformly at random in the IPv4 address ring. But we are working on the *pseudo-random full cycle* of addresses, on which the vulnerable hosts are randomly distributed.

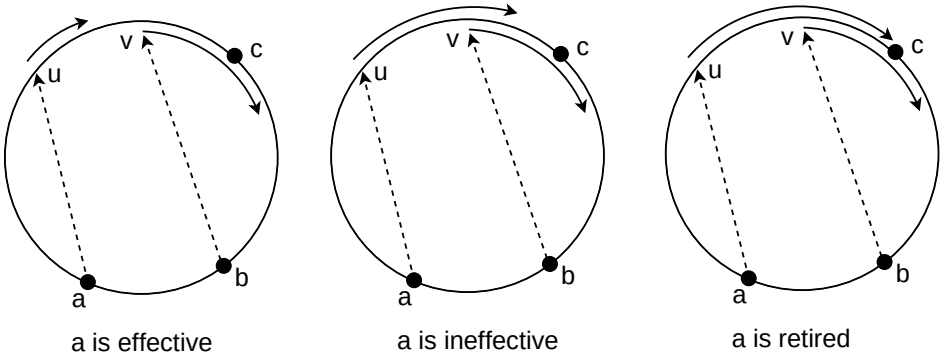

Figure 5-2. Different stages of an infected host for a full-cycle worm. An active host a is either effective or ineffective. It retires when reaching an infected host.

By time t, the combined scanning effort of all infected hosts has discovered and then infected $i(t)V-v$ vulnerable hosts. Due to random placement of vulnerable hosts, the same proportion of the $(N-v)$ addresses (which exclude the v initially infected ones) is expected to have been scanned in order to discover those hosts. Hence, the fraction $f(t)$ of the full cycle that has been scanned by time t is

$$f(t) = \frac{\frac{i(t)V-v}{V-v} \times (N-v) + v}{N} \tag{5–5}$$

Here we treat the initially infected hosts as having been scanned.

As illustrated in Fig. 5-2, when an active host a is scanning addresses that have not been scanned yet (such as $[u, v]$), it is called an *effective* host. When a is scanning addresses that have already been scanned (such as $[v, c]$), it is called an *ineffective* host. An ineffective host will become a retired host once it hits an infected host (c in this example). Let $x(t)$ be the fraction of vulnerable hosts that are effective at time t. It is worth noting that the defition of effective hosts here is slightly different from that in the modeling of Permutation-scanning worm. For the Permutation-scanning worms, a host is ineffective if its starting scanning address falls within a covered area. For full-cycle, a host is ineffective if its scanning address has been scanned before.

Because there are $x(t)V$ effective hosts at time t, all addressed having been scanned by them will form $x(t)V$ non-overlapping segments. For example, in Fig. 5-3, b and c

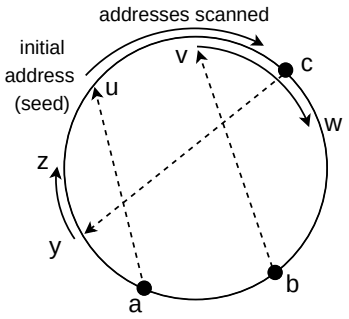

Figure 5-3. Classification of active hosts for a full-cycle worm. Hosts b and c are effective, whereas host a is ineffective because it is scanning addresses that have been scanned by b. There are two non-overlapping segments, $[u, w]$ and $[y, z]$.

are effective hosts, and there are two non-overlap segments. Host a is ineffective; its segment is merged with b's into a single bigger segment $[u, w]$. The combined size of all non-overlapping segments is $f(t)N$. Excluding the starting address, all other addresses of a segment form the *interior* of the segment. For example, the interior of $[u, w]$ is $(u, w]$. The combined size of the interior of all segments is $f(t)N - x(t)V$.

It is easy to see that, whenever the *first address* scanned by an infected host falls in the interior of a segment, there must be an effective host turned ineffective. For example, in Fig. 5-3, as host a's segment merges with host b's segment into a single non-overlapping one $[u, w]$, the first address scanned by b, which is v, becomes an address in the interior of segment $[u, w]$, and at the mean time one effective host (a in this example) becomes ineffective.

Now, because the first addresses scanned by $i(t)V$ infected hosts are randomly placed in the full cycle, the chance for each of them to be in the interior of some non-overlapping segment is $\frac{f(t)N-x(t)V}{N}$, which is the fraction of the full cycle that the interior of all non-overlapping segments occupies. The number of infected hosts having turned ineffective is $(i(t) - x(t))V$. Hence, we have

$$\begin{aligned}(i(t) - x(t))V &= \frac{f(t)N - x(t)V}{N} i(t)V \\ x(t) &= \frac{(1 - f(t))N}{N - i(t)V} i(t)\end{aligned} \tag{5–6}$$

During an indefinitely small period dt after time t, the number of scan messages sent by all effective hosts is

$$n'(t) = x(t)Vr(t)dt = \frac{(1-f(t))N}{N-i(t)V} i(t)Vrdt \tag{5–7}$$

Each scan message extends a segment by one additional address. There are two possibilities for this address. Case 1: it is located in the gap between two non-overlapping segments; Case 2: it is not located in the gap, i.e., it is the first address of the next segment on the cycle. The combined size of all gaps between non-overlapping segments is $(1-f(t))N$, and we have known that the number of non-overlapping segments is $x(t)V$. The probability for Case 1 is $\frac{(1-f(t))N}{(1-f(t))N+x(t)V}$, and since the address is not scanned before, the conditional probability for the host on the address to be a new infection is p_0 in (5–4). The probability for Case 2 is $\frac{x(t)V}{(1-f(t))N+x(t)V}$, and the conditional probability for the address (which has been scanned before) to be a new infection is zero.

Combining the above analysis, the probability for a scan message from an effective host to make a new infection is

$$\begin{aligned} p'(t) &= \frac{(1-f(t))N}{(1-f(t))N+x(t)V} \times p_0 + \frac{x(t)V}{(1-f(t))N+x(t)V} \times 0 \\ &= \frac{V-v}{N-v} \times \frac{N-i(t)V}{N} \end{aligned} \tag{5–8}$$

During the period dt, the number of newly infected hosts, $Vdi(t)$, is equal to $n'(t) \cdot p'(t)$. Applying (5–7), (5–8) and (5–5) to the equation $Vdi(t) = n'(t) \cdot p'(t)$, we have

$$di(t) = \frac{rV}{N} i(t)(1-i(t))dt$$

It can be rewritten as the following differential equation.

$$\frac{di(t)}{dt} = \frac{rV}{N} i(t)(1-i(t)) \tag{5–9}$$

It is the same as (5–3), the propagation model for ideal random-scanning worms! Because the propagation of all these worms will respond to the change in r in the same way, rate-limit techniques [77, 6] that were designed to slow down random-scanning worms

or containment techniques [62] based on worm's scanning rate will work equally well for full-cycle worms.

Solving the equation, we have

$$i(t) = \frac{e^{r\frac{V}{N}(t-t_0)}}{1 + e^{r\frac{V}{N}(t-t_0)}} \tag{5–10}$$

Since $i(0) = v/V$, applying it to (5–10), we have $t_0 = -\frac{N}{rV} \ln \frac{v}{V-v}$. Eq. (5–10) can be written as

$$i(t) = \frac{e^{r\frac{V}{N}(t+\frac{N}{rV} \ln \frac{v}{V-v})}}{1 + e^{r\frac{V}{N}(t+\frac{N}{rV} \ln \frac{v}{V-v})}} \tag{5–11}$$

From (5–11), the time it takes for a percentage α ($\geq v/V$) of all vulnerable hosts to be infected is

$$t(\alpha) = \frac{N}{r \cdot V}(\ln \frac{\alpha}{1-\alpha} - \ln \frac{v}{V-v}) \tag{5–12}$$

5.3.2 Explanation

It may look surprising that the propagation model of a full-cycle worm is identical to that of an ideal random scanning worm even though over time the full-cycle worm commits a smaller number of infected hosts to scan the Internet. We give an intuitive explanation on why this happens.

The instantaneous increase in the number of infections, $V di(t)$, is the product of two factors: the number of scan messages during dt and the probability for each scan message to generate a new infection, where the former is $n(t)$ defined in (5–1) for random-scanning worms and $n'(t)$ defined in (5–7) for full-cycle worms, while the latter is $p(t)$ defined in (5–2) for random-scanning worms and $p'(t)$ defined in (5–8) for full-cycle worms.

On one hand, by allowing more and more infected hosts to retire, full-cycle worms achieve stealth at the expense of fewer scan messages, which can be shown by comparing (5–7) and (5–1). This has a negative impact on its propagation speed.

On the other hand, random-scanning worms send scan messages to arbitrary addresses, including those that have already been scanned, which leads to lower probability

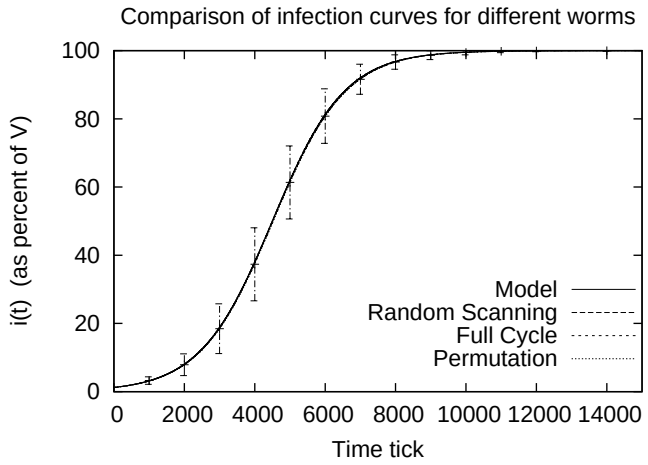

Figure 5-4. Comparison of infection curves between random-scanning, permutation-scanning and full-cycle worms. The "Model" curve is computed from (5–11). The other three curves are plotted using data collected from packet-level simulation programs that simulate worm's actual scanning behavior. These curves for "Random Scanning" worm, "Full Cycle" worm and "Permutation" worm (to be discussed in Section 5.3.4) are the average of 1,000 simulation runs. The 99% confidence interval for the curve of "Full Cycle" is plotted; the confidence intervals for "Random Scanning" and "Permutation" are comparable. All four curves in the figure completely overlap. We stress that data points calculated from (5–11) agree with data points independently collected from programs.

for each scan message to find a new infection. Full-cycle worms send scan messages mostly to addresses that have not been scanned, which means higher probability of finding new infections. This is evident by comparing (5–8) and (5–2), and it has a positive impact on the propagation speed of full-cycle worms.

Based on our mathematical calculation, interestingly, the negative impact and the positive impact exactly cancel out each other. Consequently, these worms all have the same propagation model.

5.3.3 Simulation Verification

As explained in Fig. 5-4, our simulations confirm that full-cycle worms propagate at the same speed as ideal random-scanning worms. The simulation parameters are given as follows: The size of the vulnerable population is $V = 2^{13}$. The size of the address space is $N = 2^{23}$ (it will take prohibitively long time if N is chosen to be 2^{32}). To produce an

infection curve in any of the figures in the dissertation, we simulate worm propagation for 1000 times under different random seeds, and then take the average. We normalize the time tick to be $\frac{1}{r}$; hence, an infected host sends one scan message per tick. Therefore, the same infection curves can be used for all scanning rates. For full-cycle or permutation worms, the simulation stops when all infected hosts retire. For random-scanning worms, we set a timer for the simulation to stop. For further details of the simulator, please refer to Section 4.3.4.

5.3.4 Equivalence to Permutation Worm

A permutation worm [63] maps the IP address space to a permuted space by an encryption algorithm. The corresponding decryption algorithm will map the permuted space back to the original IP space. Starting from a random location, each infected host scans continuous addresses in the permuted space. A permuted address is first decrypted to an address in the IP space, to which the scan packet will be actually sent.

Full-cycle worms are functionally equivalent to the permutation worm in the sense that it permutes the IP address space directly via the random number generator (which is needed in most known worms), instead of using encryption. Such equivalence is confirmed by our simulation (Fig. 5-4). Essentially it means that all existing random-scanning worms can be easily modified to be as powerful as permutation worms simply by using a full-cycle random number generator and incorporating a termination condition. Moreover, they have the advantage of avoiding the need to carry a decryption routine that would increase the length of the worm code substantially and make it more susceptible to detection [67].

When an active host of a full-cycle worm scans an already infected host, instead of letting it retire immediately, we may modify the worm and allow the host to jump to a new random location in the cycle for its new scan target and resume its scanning from there. The host will retire after a certain number of jumps. We performed simulations to evaluate how jumps affect propagation speed and scanning traffic volume. Our simulation results are identical to results shown in Fig. 4-7 for permutation worm, which shows that

jumps by infected hosts only modestly increase the propagation speed of a full-cycle worm, at the cost of considerable increase in the scanning traffic volume, hurting stealthiness.

5.4 Stealthiness of Full-Cycle Worms

In this section, we derive the closed-form solutions for $x(t)$, $a(t)$ and $s(t)$, which are the fractions of vulnerable hosts that are effective, active and retired, respectively. Based on these solutions, we know the number of effective hosts $x(t)V$, the number of active hosts $a(t)V$, and the number of retired hosts $s(t)V$ over time.

The results will help us assess the stealthiness of full-cycle worms. For example, we learn that irrespective of v, the maximum instantaneous footprint of these worms, which is the largest instantaneous scanning rate by all infected hosts (i.e., $\max_t\{a(t)Vr\}$), is about 43% of the value for random-scanning worms. We also learn that the gross footprint of full-cycle worms is within a factor of 1.5 from the optimal.

5.4.1 Number of Effective Hosts over Time

Applying (5–5) and (5–10) to (5–6), we have the closed-form solution for $x(t)$.

$$x(t) = \frac{(1 - \frac{\frac{i(t)V-v}{V-v}\times(N-v)+v}{N})N}{N - i(t)V} \times \frac{e^{r\frac{V}{N}(t+\frac{N}{r\cdot V}\ln\frac{v}{V-v})}}{1 + e^{r\frac{V}{N}(t+\frac{N}{r\cdot V}\ln\frac{v}{V-v})}} \tag{5–13}$$

Recall that the number of effective hosts is $x(t)V$. Eq (5–13) is hard to interpret. We perform approximation below. Suppose $v \ll V$ (and obviously, $v \ll N$). Eq (5–5) can be simplified to

$$f(t) \approx i(t). \tag{5–14}$$

Applying it to (5–6), we have

$$x(t) \approx \frac{(1 - i(t))N}{N - i(t)V} i(t). \tag{5–15}$$

Suppose $V \ll N$, which is likely to hold as the number of hosts vulnerable to a particular worm normally accounts for a small portion of the whole Internet address space. The above equation can be further simplified to

$$x(t) \approx (1 - i(t))i(t). \tag{5–16}$$

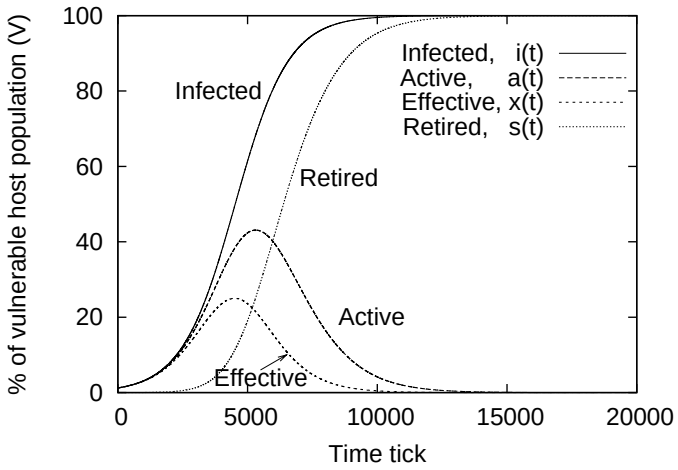

Figure 5-5. Simulation results on full-cycle worm propagation

It follows that

$$\max_t x(t) \approx \frac{1}{4} \tag{5–17}$$

The maximum value of $x(t)$ is reached at approximately the time when $i(t) = 1 - i(t) = \frac{1}{2}$, namely, when half of the vulnerable hosts are infected, which is confirmed by our simulation results in Fig. 5-5 (see Section 5.3.3 for simulation setup). By (5–12), that time is

$$t(\frac{1}{2}) = \frac{N}{rV}(\ln(V - v) - \ln v) \approx \frac{N \ln V}{rV} \tag{5–18}$$

5.4.2 Number of Active Hosts

We have already obtained the closed-form solution for $i(t)$ in (5–11). Below we obtain the closed-form solution for $a(t)$. Once $a(t)$ is known, by $i(t) = a(t) + s(t)$, we also get the closed-form solution for $s(t)$.

Throughout this and next subsection, we use the notation $\phi = \frac{v}{V} = i(0) = a(0) =$ fraction of vulnerable host population already infected at time 0. Using this ϕ in (5–11), we can rewrite the closed form for $i(t)$ as

$$i(t) \quad = \quad \frac{\phi e^{\frac{rV}{N}t}}{1 - \phi + \phi e^{\frac{rV}{N}t}} \tag{5–19}$$

Since each active host has $\frac{V}{N}$ chance for hitting a vulnerable host for each scan message, in time dt the expected number of hits on vulnerable hosts, including both previously uninfected and previously infected, is $a(t)V \times r \times dt \times \frac{V}{N}$. We note that when an active host hits a previously uninfected host, it adds to the overall infection count $i(t)$. Since $i(t)$ changes by $di(t)$ within time dt, the total number of hits on previously uninfected vulnerable hosts must be $di(t)$. On the other hand, when an active host hits an already infected host, it retires and thereby adds to the $s(t)$ count. Since $s(t)$ changes by $ds(t)$ within time dt, the total number of hits on previously uninfected vulnerable hosts must be $ds(t)$. Thus, we get $a(t)V \times r \times dt \times \frac{V}{N}$ = total number of hits = $di(t) + ds(t)$. In other words, $ds(t) = a(t)V \times r \times dt \times \frac{V}{N} - di(t)$. Since $da(t) = di(t) - ds(t)$, we also get $da(t) = 2di(t) - a(t)V \times r \times dt \times \frac{V}{N}$. Plugging back $di(t) = r \times dt \times \frac{V}{N} i(t)(1 - i(t))$, we get the final propagation equations:

$$\frac{di(t)}{dt} = \frac{rV}{N} \times i(t) \times (1 - i(t)) \tag{5–20}$$

$$\frac{da(t)}{dt} = \frac{rV}{N} \times (2\, i(t) \times (1 - i(t)) - a(t)) \tag{5–21}$$

$$\frac{ds(t)}{dt} = \frac{rV}{N} \times (a(t) - i(t) \times (1 - i(t))) \tag{5–22}$$

Applying (5–19) to (5–21), we have

$$\frac{da(t)}{dt} = \frac{rV}{N} \times \left(\frac{2(1-\phi)\phi e^{\frac{rV}{N}t}}{(1 - \phi + \phi e^{\frac{rV}{N}t})^2} - a(t) \right) \tag{5–23}$$

Solving the above equation for $a(t)$ using the boundary condition of $a(0) = \phi$, and based on $i(t) = a(t) + s(t)$, we have the following closed forms for infected, active and retired

fraction of vulnerable hosts at time t:

$$i(t) = \frac{\phi e^{\frac{rV}{N}t}}{1-\phi+\phi e^{\frac{rV}{N}t}} \tag{5–24}$$

$$a(t) = \frac{2(1-\phi)}{\phi e^{\frac{rV}{N}t}}\left\{\frac{1-\phi}{1-\phi+\phi e^{\frac{rV}{N}t}} - 1 + \phi + \ln(1-\phi+\phi e^{\frac{rV}{N}t}) + \frac{\phi^2}{2(1-\phi)}\right\} \tag{5–25}$$

$$s(t) = \frac{\phi e^{\frac{rV}{N}t}}{1-\phi+\phi e^{\frac{rV}{N}t}} - \frac{2(1-\phi)}{\phi e^{\frac{rV}{N}t}}\left\{\frac{1-\phi}{1-\phi+\phi e^{\frac{rV}{N}t}} - 1 + \phi + \ln(1-\phi+\phi e^{\frac{rV}{N}t}) + \frac{\phi^2}{2(1-\phi)}\right\} \tag{5–26}$$

We observe that the differential equations 5–20 through 5–22 are the same as equations 4–22 through 4–24, respectively. Therefore, it comes as no surprise that closed-form solutions (equations 5–24 through 5–26) are the same as equations 4–25 through 4–27, respectively.

5.4.3 Maximum Instantaneous Footprint (Peak Scanning Traffic)

The maximum instantaneous footprint is defined as the maximum combined rate of scanning by all infected hosts at any time instant. For a random-scanning worm, it is Vr, which happens when all V vulnerable hosts are infected. For a full-cycle worm, because of retirement, it is $\max_t\{a(t)Vr\}$. Therefore, the maximum instantaneous footprint is determined by the maximum value of $a(t)$ over time t. We will show that $\max_t\{a(t)\}$ is around 43% for ϕ from 0.01% up to 30%.

$a(t)$ reaches its maximum value when $\frac{da(t)}{dt} = 0$. Setting the right-hand side of (5–21) to zero, we have

$$\max_t a(t) = 2\, i(t)(1-i(t)) \tag{5–27}$$

Applying (4–25), we have

$$\max_t a(t) = \frac{2(1-\phi)\phi e^{\frac{rV}{N}t}}{(1-\phi+\phi e^{\frac{rV}{N}t})^2} \tag{5–28}$$

Since $a(t)$ is given by (4–26), we observe that, at time t where the maximum value of $a(t)$ is reached, the following is true,

$$\max_t a(t) = \frac{2RY}{(R+Y)^2} = \frac{2R}{Y}\left\{\frac{R}{R+Y} - R + \ln(R+Y) + \frac{\phi^2}{2R}\right\}$$

where $R = 1-\phi$ and $Y = \phi e^{\frac{rV}{N}t}$ for simplification. Let $g = \frac{Y}{R} = \frac{\phi e^{\frac{rV}{N}t}}{1-\phi}$. Then, $\max_t a(t) = \frac{2g}{(1+g)^2}$. Substituting $Y = gR$ into the above equation and simplifying, we get the following:

$$\frac{g^2}{(1+g)^2} = \frac{1}{1+g} - R + \ln(R) + \ln(1+g) + \frac{\phi^2}{2R}$$

Bringing all the g-terms to the left-hand side, and replacing $R = 1-\phi$ back, we have

$$\frac{g^2}{(1+g)^2} + \frac{g}{1+g} - \ln(1+g) \quad = \quad \Delta(\phi) \tag{5–29}$$

where $\Delta(\phi) = \phi + \ln(1-\phi) + \frac{\phi^2}{2(1-\phi)}$. Using Taylor series expansion, $\Delta(\phi) = \frac{\phi^3}{6} + \frac{\phi^4}{4} + \frac{3\phi^5}{10} + \frac{\phi^6}{3} + ...$

Now, if $\Delta(\phi)$ in (5–29) were to be a constant, then it would be seen immediately that only a constant value of g can satisfy that equation. In that case, $\max_t a(t) = \frac{2g}{(1+g)^2}$ would again be a constant. To test our hypothesis that the value of $\Delta(\phi)$ is indeed almost a constant, we take different values of ϕ and see how the values of $\Delta(\phi)$, g and $\max_t a(t)$ change along with it. The results are shown in the following table.

Table 5-1. Effect of hitlist size on the scanning peak.

ϕ	$\Delta(\phi)$	g	$\max_t a(t)$
0.01 %	0.000000	2.162580	43.2 %
0.1 %	0.000000	2.162580	43.2 %
1 %	0.000000	2.162580	43.2 %
2 %	0.000001	2.162570	43.2 %
5 %	0.000022	2.162300	43.2 %
10 %	0.000195	2.160130	43.2 %
20 %	0.001856	2.139130	43.4 %
30 %	0.007611	2.064960	43.9 %

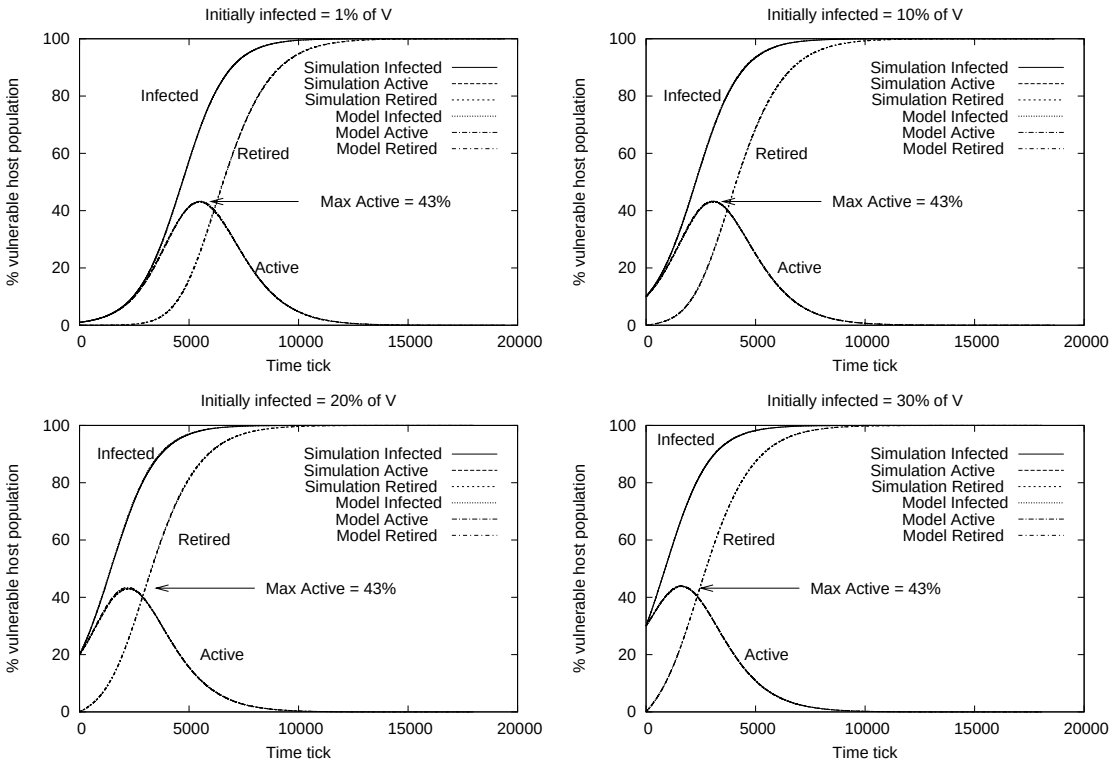

Figure 5-6. Propagation patterns for the full-cycle worm with different ϕ. In all the cases, the peak scanning volume is reached when around 43% of the vulnerable population are actively scanning. Also, in all cases, the results from our worm simulator matches the result from the model with amazing precision (the two sets of curves cannot be distinguished unless viewed at a very high magnification (around 64X).

In all cases $\Delta(\phi)$ is almost 0, which is understandable since $\Delta(\phi) = \Theta(\phi^3)$ as seen above. Therefore, we can say that for reasonable values of initially infected host ratio ϕ, $\max_t a(t)$ is reached around 43%. We verify this fact by both simulation and model output in Figure 5-6.

5.4.4 Gross Footprint

The total number of scan messages by effective hosts is N. The expected number of addresses scanned by an ineffective host is $\frac{N}{2V}$. Each infected host will become ineffective for once. Therefore, the expected number of addresses scanned by all ineffective hosts over time is $\frac{N}{2V}V = \frac{N}{2}$. The gross footprint of a full-cycle (or permutation) worm, defined as the overall scan volume, is thus $N + \frac{N}{2} = \frac{3}{2}N$. Hence, each address is scanned 1.5

times on average. In comparison, because existing random-scanning worms do not have a termination condition, their gross footprint is not bounded. Hence the worm defense techniques targeting at high scan volume of random-scanning worms will be less effective when applied against full-cycle worms.

5.5 Quest for the Optimal Strategy

The virulence of a worm is indicated by how quickly, resiliently and stealthily it can comprise the entire vulnerable host population, and it is dependent on the scanning strategy chosen by the worm. Before we delve into the search for an optimal scanning strategy, it is to be noted that we use the word "optimal" in a rather loose and subjective way rather than maximizing some objective function. The reason for this choice is the absence of any universally accepted standard for measuring the effectiveness of a scanning strategy. Nevertheless, we observe that a worm employing an optimal scanning strategy must have the following characteristics:

- **Infection speed:** The worm should be able to infect the entire vulnerable host population as quickly as possible. Pictorially, the slope of the infection curve (plot of $i(t)$ over time) should be as steep as possible.
- **Stealth:** It should not have a high network footprint. In other words, the area under the active curve (plot of $a(t)$ over time) should be as less as possible.
- **Fault tolerance:** Even if a portion of the infected host population are removed or patched, the worm should still be able to infect the entire vulnerable population.

We show that the scanning strategy used by the full-cycle worm meets all three of the criteria above. First, we observe that it has the identical infection speed as the random-scanning worm. However, for a random-scanning worm, all the infected hosts are scanning at the same time, which means that the infection curve and the active curve are one and the same. On the other hand, for a full-cycle worm, irrespective of the hitlist size, the active curve reaches a scanning peak around 43% of the vulnerable population size and then drops towards zero. In comparison with the random-scanning worm, this ensures a much smaller network footprint, which has been estimated as $\frac{3}{2}N$. While a

full-cycle worms scans each address 1.5 times on the average, there is no such bound for existing random-scanning worms without a termination condition. Thus, full-cycle worms are significantly more stealthy than ordinary random-scanning worms. Finally, since the propagation strategy in full-cycle worm allows random jumps, even if an active hosts is removed from the vulnerable pool, some other host will eventually make a jump in the vicinity of the address that was last scanned by the original host, and complete the unfinished scanning work. This way, the full-cycle worm enjoys good fault tolerance property. In essence, since the full-cycle worm possesses all the three desired properties described above, we conclude that it is optimal.

5.6 Contributions

This is the first work that studies the impact of pseudo randomness on Internet worm propagation. We give a closed-form solution to the modeling problem of full-cycle (and permutation) worms. The analytical results provide a number of interesting insights into these worms. Notably, while allowing infected hosts to retire, these worms' infection curve is identical to that of random-scanning worms, even through they send much less scan messages. We formally analyze the stealthiness of full-cycle worms in terms of maximum instantaneous footprint and gross footprint. Finally, we show that a full-cycle worm enjoys the properties desired by an optimal scanning strategy.

CHAPTER 6
CONCLUSIONS

Our study undertook three important problems: worm detection, worm propagation modeling, and worm design. First, we devised a fast and reliable detection mechanism for the ASCII worms and verified its efficiency through experiments. We also derived the statistical model of the maximum executable length (MEL) scheme underlying our detection mechanism, which serves as the foundation of not only our detection mechanism but also several other similar MEL-based detection mechanisms that set the MEL threshold experimentally. Our mathematical model also established the relation between the MEL threshold and the false positive error probability, which means our analysis makes it possible to tune the detection sensitivity of any MEL-based scheme.

We also derived the propagation model for the permutation-scanning worm, and through extensive simulations, have shown that the analytical model is accurate. We have extended our model for permutation-scanning worms employing multiple jumps, and obtained perfect match between the output of the mathematical model and the worm simulations.

Finally, we examined the role of pseudo-randomness in the propagation for worms using a random number generator and discovered the flaws in the derivation of the classical epidemic model, which till date has served as the universally accepted model for propagation of random-scanning worms. At the same time, we have also discovered that we can exploit this pseudo-randomness to our favor. By using a specific pseudo-random number generator and incorporating a termination criterion, we have shown that existing random scanning worms can be made significantly stealthier without losing any infection speed, thereby making this particular scanning strategy a very efficient one.

Overall, our work focused on highlighting the damage potential of worms, and showed novel ways to detect them. It also provided accurate analytical propagation model

for worms. This can help network security personnel to better understand the worms' spreading behavior, and design containment techniques and other countermeasures.

REFERENCES

[1] The Metasploit Project. Online Text. *http://www.metasploit.com/*, Copyright © 2003-2008 Metasploit™ LLC, Austin, Texas, USA.

[2] P. Akritidis, E.P. Markatos, M. Polychronakis, and K. Anagnostakis. Stride: Polymorphic Sled Detection through Instruction Sequence Analysis. In *Proc. of the* 20^{th} *IFIP International Information Security Conference*, May 2005.

[3] Aleph One. Smashing the Stack for Fun and Profit. *Phrack*, 7(49), November 1996.

[4] S. Bhatkar, R. Sekar, and D.C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proc. of the* 14^{th} *USENIX Security Symposium*, July 2005.

[5] ByteEnable. Linux Kernel Now With AMD64 x86 NX (No eXecute) Bit Support. Online Text. *http://www.linuxelectrons.com/news/linux/linux-kernel-now-amd64-x86-nx-no-execute-bit-support*, June 2004, Copyright © 2003 – 2008 LinuxElectrons™, Cedar Park, Texas 78630, USA.

[6] S. Chen and Y. Tang. Slowing Down Internet Worms. In *Proc. of the 24th International Conference on Distributed Computing Systems (ICDCS '04)*, Tokyo, Japan, March 2004.

[7] Z. Chen, L. Gao, and K. Kwiat. Modeling the Spread of Active Worms. In *Proc. of the* 22^{nd} *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '03)*, pages 1890–1900, San Francisco, California, USA, March 2003.

[8] Z. Chen and C. Ji. Measuring Network-Aware Worm Spreading Ability. In *Proc. of the* 26^{th} *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '07)*, May 2007.

[9] R. Chinchani and E. V. D. Berg. A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows. In *Proc. of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID '05)*, September 2005.

[10] Computer Emergency Response Team. CERT® Advisory CA-2000-04 Love Letter Worm. Online Text. *http://www.cert.org/advisories/CA-2000-04.html*, May 2000, Copyright © 2000 Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

[11] Computer Emergency Response Team. CERT® Advisory CA-2001-26 Nimda Worm. Online Text. *http://www.cert.org/advisories/CA-2001-26.html*, September 2001, Copyright © 2001 Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

[12] Computer Emergency Response Team. CERT® Advisory CA-1999-04 Melissa Macro Virus. Online Text. *http://www.cert.org/advisories/CA-1999-04.html*, March 2003, Copyright © 1999 Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

[13] Computer Emergency Response Team. CERT® Advisory CA-2001-04: MS-SQL Server Worm. Online Text. http://www.cert.org/advisories/CA-2003-04.html, January 2003, Copyright © 2003 Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

[14] Computer Emergency Response Team. CERT® Incident Note IN-2003-03 W32/Sobig.F Worm. Online Text. http://www.cert.org/incident_notes/IN-2003-03.html, August 2003, Copyright © 2003 Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

[15] Computer Emergency Response Team. Technical Cyber Security Alert TA04-028A W32/MyDoom.B Virus. Online Text. http://www.us-cert.gov/cas/techalerts/TA04-028A.html, August 2004, Copyright © 2004 Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

[16] A. Corlett, D. I. Pullin, and S. Sargood. Statistics of One-Way Internet Packet Delays. In *Proc. of IETF 2002*, March 2002.

[17] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end Containment of Internet Worms. In *Proc. of the* 20^{th} *ACM SOSP*, October 2005.

[18] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proc. of the* 12^{th} *conference on USENIX Security Symposium (SSYM'03)*, pages 91–104, Berkeley, California, USA, 2003. USENIX Association.

[19] 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 *Proc. of the* 7^{th} *USENIX Security Conference (Security '98)*, pages 63–78, January 1998.

[20] J.R. Crandall, S.F. Wu, and F.T. Chong. Experiences Using Minos as a Tool for Capturing and Analyzing Novel Worms for Unknown Vulnerabilities. In *Proc. of Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, July 2005.

[21] D. Knowles and F. Perriott, Symantec Security Response. W32.blaster.worm. Online Text. http://www.symantec.com/security_response/writeup.jsp?docid=2003-081113-0229-99\&tabid=2, August 2003, Copyright © 1995 – 2008 Symantec Corporation, 20330 Stevens Creek Blvd. Cupertino, CA 95014, USA.

[22] D. Moore and C. Shannon and K. Claffy. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *Proc. of the* 2^{nd} *Internet Measurement Workshop (IMW '02)*, November 2002.

[23] P. J. Denning. *Computers under Attack: Intruders, Worms, and Viruses.* ACM Press, New York, NY, USA, 1990.

[24] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. V. Underduk. Polymorphic Shellcode Engine using Spectrum Analysis. *Phrack*, 11(61), August 2003.

[25] Y. Dong and K. David. Reliable Return Address Stack: Microarchitectural Features to Defeat Stack Smashing. In *Proc. of ACM SIGARCH Computer Architecture News*, volume 33-1, pages 73–80, March 2005.

[26] R. Eller. Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel platforms. Online Text. *http://www.task.to/forum/viewtopic.php?t=131*, 2003, Copyright © 2004-2008 Toronto Area Security Klatch, Toronto, Canada.

[27] F. Perriot, Symantec Security Response. W32.welchia.worm. Online Text. *http://www.symantec.com/security_response/writeup.jsp?docid=2003-081815-2308-99\&tabid=2*, August 2003, Copyright © 1995 – 2008 Symantec Corporation, 20330 Stevens Creek Blvd. Cupertino, CA 95014, USA.

[28] J. C. Frauenthal. Mathematical Modeling in Epidemiology. *Springer-Verlag, New York*, 1980.

[29] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *Proc. of IEEE Symposium on Security and Privacy, 2004*, pages 211–225, May 2004.

[30] H. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proc. of the* 13^{th} *USENIX Security Symposium (Security '04)*, pages 271–286, San Diego, California, USA, August 2004.

[31] D. E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, third edition, 1997.

[32] O. Kolesnikov and W. Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. Technical report, College of Computing, Georgia Institute of Technology, 2004.

[33] C. Kreibich and J. Crowcroft. Honeycomb – Creating Intrusion Detection Signatures using Honeypots. In *Proc. of the* 2^{nd} *Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, Massachusetts, USA, November 2003.

[34] C. Kruegel, E. Kirda, D. Mutz W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Proc. of the* 8^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID '05)*, September 2005.

[35] A. Kumar, V. Paxson, and N. Weaver. Exploiting Underlying Structure for Detailed Reconstruction of an Internet-scale Event. In *Proc. of Internet Measurement Conference*, October 2005.

[36] Kaspersky Lab. Virus Encyclopedia. Online Text. *http://www.viruslist.com/en/viruses/encyclopedia*, Copyright © 1996 – 2008, Kaspersky Lab, 500 Unicorn Park, 3rd Floor Woburn, MA 01801, USA.

[37] Z. Li, M. Sanghi, Y. Chen, M. Kao, and B. Chavez. Hamsa: Fast Signature Generation for Zero-day Polymorphic Worms with Provable Attack Resilience. In *Proc. of IEEE Symposium on Security and Privacy*, May 2006.

[38] Z. Li, L. Wang, Y. Chen, and Z. Fu. Network-based and Attack-resilient Length Signature Generation for Zero-day Polymorphic Worms. In *Proc. of the 15th IEEE International Conference on Network Protocols (ICNP)*, October 2007.

[39] J. Ma, G. M. Voelker, and S. Savage. Self-Stopping Worms. In *Proc. of the 2005 ACM workshop on Rapid malcode (WORM '05)*, pages 12–21, New York, NY, USA, November 2005. ACM.

[40] P. K. Manna, S. Chen, and S. Ranka. Exact Modeling of Propagation for Permutation-Scanning Worms. In *Proc. of the* 27^{th} *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '08)*, April 2008.

[41] P. K. Manna, S. Ranka, and S. Chen. DAWN: A Novel Strategy for Detecting ASCII Worms in Networks. In *Proc. of the* 27^{th} *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '08) mini-conference*, April 2008.

[42] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

[43] D. Moore, C. Shannon, G. M. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proc. of the* 22^{nd} *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '03)*, pages 1901–1910, San Francisco, California, USA, March 2003.

[44] J. Newsome, B. Karp, and D. Song. Polygraph: Automatic Signature Generation for Polymorphic Worms. In *Proc. of IEEE Security and Privacy Symposium*, Oakland, California, USA, May 2005.

[45] Oxford University Press. FAQ: Ask Oxford. Online Text. *http://www.askoxford.com/asktheexperts/faq/aboutwords/frequency?view=uk*, Copyright © 2008 Oxford University Press, Great Clarendon Street, Oxford, OX2 6DP, UK.

[46] P. Szor, Symantec Security Response. Freebsd.scalper.worm. Online Text. *http://www.symantec.com/security_response/writeup.jsp?docid=2002-062814-5031-99*, June 2002, Copyright © 1995 – 2008 Symantec Corporation, 20330 Stevens Creek Blvd. Cupertino, CA 95014, USA.

[47] K. Park, H. Kim, B. Bethala, and A. Selcuk. Scalable Protection against DDoS and Worm Attacks. *DARPA ATO FTN, Technical Report AFRL-IF-RS-TR-2004-100, Dept. of Computer Science, Purdue University*, 2004.

[48] S. K. Park and K. W. Miller. Random Number Generators: Good Ones Are Hard to Find. *Communications of the ACM*, 31(10), March 1988.

[49] Y.J. Park and G. Lee. Repairing Return Address Stack for Buffer Overflow Protection. In *Proc. of ACM Frontiers of Computing*, pages 335–342, April 2004.

[50] A. Pasupulati, J. Coit, K. Levitt, and F. Wu. Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities. In *Proc. of IEEE/IFIP Network Operation and Management Symposium*, May 2004.

[51] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Network-Level Polymorphic Shellcode Detection Using Emulation. In *Proc. of the GI/IEEE SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, pages 54–73. Springer, 2006.

[52] Charles Price. MIPS IV Instruction Set, Revision 3.2. Online Text. *http://math-atlas.sourceforge.net/devel/assembly/mips-iv.pdf*, September 1995, Copyright © 1995 MIPS Technologies, Inc. 1225 Charleston Rd. Mountain View, California 94043, USA.

[53] X. Qin, D. Dagon, G. Gu, and a Lee. Worm Detection Using Local Networks. In *Proc. of 20th Annual Computer Security Applications Conf. (ACSAC 2004)*, December 2004.

[54] R. X. Wang, Symantec Security Response. W32.zotob.a. Online Text. *http://www.symantec.com/security_response/writeup.jsp?docid=2005-081415-0646-99*, August 2005, Copyright © 1995 – 2008 Symantec Corporation, 20330 Stevens Creek Blvd. Cupertino, CA 95014, USA.

[55] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, and R. K. Cunningham. Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code. In *Proc. of the 2003 ACM Workshop on Rapid Malcode*, October 2003.

[56] RIX. Writing IA32 Alphanumeric Shellcodes. *Phrack*, 11(57), November 2001.

[57] J. A. Rochlis and M. W. Eichin. With Microscope and Tweezers: The Worm from MIT's Perspective. *Commun. ACM*, 32(6):689–698, 1989.

[58] S. Schechter, J. Jung, and A. W. Berger. Fast Detection of Scanning Worm Infections. In *Proc. of the 7^{th} International Symposium on Recent Advances in Intrusion Detection (RAID '04)*, Sophia Antipolis, French Riviera, France, September 2004.

[59] E. E. Schultz. Where Have the Worms and Viruses Gone? New Trends in Malware. *Computer Fraud & Security*, 2006(7):4–8, August 2006.

[60] SecurityFocus. A Zero-day Worm in IE. Online Text. *http://www.securityfocus.com/archive/1/358914/2004-03-27/2004-04-02/2*, April 2004, Copyright © 2008 SecurityFocus, Suite # 1000 100 4th Avenue S.W. Calgary, AB T2P 3N2, Canada.

[61] S. Singh, C. Estan, G. Varghese, and S. Savage. The EarlyBird System for Real-time Detection of Unknown Worms. In *Proc. of the 6^{th} ACM/USENIX Symposium on*

Operating System Design and Implementation (OSDI '04), San Francisco, California, USA, December 2004.

[62] S. Staniford. Containment of Scanning Worms in Enterprise Networks. *Journal of Computer Security*, 2004.

[63] S. Staniford, V. Paxson, and N. Weaver. How to 0wn the Internet in Your Spare Time. In *Proc. of the* 11^{th} *USENIX Security Symposium (Security '02)*, San Francisco, California, USA, August 2002.

[64] Symantec Enterprise Security. Symantec Internet Security Threat Report, Trends for July–December 07. Online Text. *http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xiii_04-2008.en-us.pdf*, April 2008, Copyright © 1995 – 2008 Symantec Corporation, 20330 Stevens Creek Blvd. Cupertino, CA 95014, USA.

[65] Y. Tang and S. Chen. Defending against Internet Worms: A Signature-Based Approach. In *Proc. of the* 24^{th} *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '04)*, March 2005.

[66] Computer Emergency Response Team. CERT® Advisory CA-2001-23 Continued Threat of the "Code Red" Worm. Online Text. *http://www.cert.org/advisories/CA-2001-23.html*, July 2001, Copyright © 2000 Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

[67] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proc. of the* 5^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID '02)*, October 2002.

[68] J. Twycross and M. M. Williamson. Implementing and Testing a Virus Throttle. In *Proc. of the* 12^{th} *USENIX Security Symposium (Security '03)*, pages 285–294, Washington D.C., USA, August 2003.

[69] Y. Ukai and D. Soeder. Analysis: Sasser Worm. Online Text. *http://research.eeye.com/html/advisories/published/AD20040501.html*, May 2004, Copyright © 1998-2008 eEye Digital Security, 111 Theory Suite 250, Irvine, CA 92617, USA.

[70] M. Vojnovic, V. Gupta, T. Karagiannis, and C. Gkantsidis. Sampling Strategies for Epidemic-Style Information Dissemination. In *Proc. of the* 27^{th} *Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '08)*, April 2008.

[71] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proc. of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '04)*, pages 193–204, Portland, Oregon, USA, August 2004. ACM Press.

[72] K. Wang and S. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proc. of the* 7^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID '04)*, September 2004.

[73] X. Wang, C. Pan, P.P. Liu, and S. Zhu. A Signature-free Buffer Overflow Attack Blocker. In *Proc. of the* 15^{th} *USENIX Security Symposium*, July 2006.

[74] D. L. Weaver and T. Germond, editors. *SPARC® Architecture Manual v9*. PTR Prentice Hall, A Paramount Communications Company, Englewood Cliffs, New Jersey 07632, USA, 1992.

[75] N. Weaver, I. Hamadeh, G. Kesidis, and V. Paxson. Preliminary Results Using Scale-down to Explore Worm Dynamics. In *Proc. of the 2004 ACM Workshop on Rapid Malcode (WORM '04)*, pages 65–72, Washington DC, USA, March 2004. ACM Press.

[76] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *Proc. of the* 13^{th} *USENIX Security Symposium (Security '04)*, pages 29–44, San Diego, California, USA, August 2004.

[77] M. M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. In *Proc. of the* 18^{th} *Annual Computer Security Applications Conference (ACSAC '02)*, pages 61–68, Las Vegas, Nevada, USA, December 2002.

[78] G. Yan and S. Eidenbenz. Modeling Propagation Dynamics of Bluetooth Worms. In *Proc. of ICDCS '07*, June 2007.

[79] A. Zeichick. Security Ahoy! Flying the NX Flag on Windows and AMD64 To Stop Attacks. *AMD Developer Central*, March 2005.

[80] C. C. Zou, W. Gong, and D. Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proc. of the* 9^{th} *ACM Conference on Computer and Communications Security (CCS '02)*, pages 138–147, Washington, DC, USA, November 2002. ACM Press.

BIOGRAPHICAL SKETCH

Parbati received his B.Tech degree from Indian Institute of Technology, Kharagpur in 1997. He obtained his M.S. in computer and information science and engineering from the University of Florida in 2007, after which he continued pursuing his Ph.D. from the same university. Between 1997 and 2002, he worked in the renowned Indian software company Infosys Technologies Ltd. He held prestigious NTSE (National Talent Search Examination) scholarship and Merit Scholarship endowed by the Government of India. His research area includes malware propagation and detection, designing malware of the future, and intrusion detection.