End-to-End Containment of Internet Worm Epidemics

Manuel Costa

Churchill College

University of Cambridge

A thesis submitted for the degree of

Doctor of Philosophy

October 2006

Declaration

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university. Further, no part of the dissertation has already been or is being concurrently submitted for any such degree, diploma or other qualification. This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text. This dissertation contains less than 60,000 words, including tables, footnotes, appendices, bibliography, and diagrams.

Abstract

Worms — programs that self-replicate automatically over computer networks — are a serious threat to hosts connected to the Internet. They infect hosts by exploiting software vulnerabilities, and they can use their victims for many malicious activities. Past outbreaks show that worms can spread too fast for humans to respond, hence worm containment must be automatic.

Recent work proposed network-level techniques to automate worm containment, but these techniques have limitations because there is no information about vulnerabilities at the network level. We propose Vigilante: a new end-to-end architecture to contain worms automatically that addresses these limitations.

In Vigilante, hosts detect worms by instrumenting vulnerable programs to analyze infection attempts. We introduce *dynamic data-flow analysis*: a broad-coverage host-based algorithm that can detect unknown worms, by tracking the flow of data from network messages, and disallowing unsafe uses of that data. We also show how to integrate other host-based detection mechanisms into the Vigilante architecture.

Upon detection, hosts generate *self-certifying alerts* (SCAs), a new type of security alert that can be inexpensively verified by any vulnerable host. Using SCAs, hosts can cooperate to contain an outbreak, without having to trust each other. Vigilante broadcasts SCAs over an overlay network that propagates alerts rapidly and resiliently.

Hosts receiving an SCA protect themselves by generating filters with *vulnerability condition slicing*: an algorithm that performs dynamic analysis of the vulnerable program to identify control-flow conditions that lead to successful attacks. These filters block the worm attack, including all mutations that follow the execution path identified by the SCA, while introducing a negligible performance overhead.

Our results show that Vigilante can contain fast spreading worms that exploit unknown vulnerabilities without false positives. Vigilante does not require any changes to hardware, compilers, operating systems or the source code of vulnerable programs; therefore, it can be used to protect software as it exists today in binary form.

Acknowledgements

I'm very grateful to my advisor, Jon Crowcroft, for accepting me as his student, and for his constant support, wise advice and enthusiasm. My co-advisors, Miguel Castro and Ant Rowstron, are a constant source of inspiration and I cannot thank them enough for helping me with this work. I feel very fortunate to have had the chance to work with Jon, Miguel and Ant.

This work would have been impossible without the help of Andrew Herbert, who supported me from the beginning, and provided an extraordinary environment at the Microsoft Research Cambridge lab. I'm very grateful to Paul Barham for supporting this work, and for many useful discussions. I thank Lidong Zhou and Lintao Zhang, for all of their help. Darek Mihocka, Sanjay Bhansali, Eric Traut, Rene Wilhelm, Henk Uijterwaal and John Wilander provided invaluable support and data for this work.

I thank John Manferdelli and Marcus Peinado, for valuable discussions and for taking this project to the next level. I thank Fred Schneider and Jed Crandall for comments that helped to improve this work.

I thank my colleagues at the Cambridge Computer Lab, Christian Kreibich, Steve Hand, Evangelia Kalyvianaki, Anil Madhavapeddy, Keir Fraser, Alex Ho, Andy Warfield, Michael Fetterman, Wenjun Hu, Eng Keong Lua, Menghow Lim, and Ian Pratt, for many useful discussions.

I thank my colleagues at Microsoft Research Cambridge, Tuomas Aura, Karthik Bhargavan, Richard Black, Lucas Bordeaux, Byron Cook, Luca Cardelli, Austin Donnelly, Cédric Fournet, Christos Gkantsidis, Ayalvadi Ganesh, Andy Gordon, Mitch Goldberg, Dinan Gunawardena, Youssef Hamadi, Tim Harris, Tony Hoare, Rebecca Isaacs, Peter Key, John Miller, Richard Mortier, Brendan Murphy, Dushyanth Narayanan, Greg O'Shea, Mike Roe, Pablo Rodriguez Rodriguez, and Milan Vojnovic, for many conversations about this work and for making the lab a great place.

I'm eternally grateful to my parents, Inácia and Manuel, and my sister, Manuela, for their support over the years. Above all, I thank my daughter, Rita, for filling my life with happiness.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Worms — programs that self-replicate automatically over computer networks — are a serious threat to computers connected to the Internet. They spread by exploiting low-level software defects, and they can use their victims for illicit activities, such as corrupting data, sending unsolicited electronic mail messages, generating traffic for distributed denial of service attacks, or stealing information. Past outbreaks show that worms can rapidly infect hundreds of thousands of computers. For instance, Slammer [MPS+03] infected 90% of all the vulnerable machines in the Internet (approximately 75,000 computers), in 10 minutes. While Slammer didn't have a particularly malicious behaviour, the consequences would have been disastrous if the worm had chosen to erase the disks of the infected machines. Incidents such as this make it clear that we cannot leave network-connected computers unprotected from worm attacks.

One avenue to deal with this problem is prevention. Since worms need to exploit software defects, by eliminating all software defects we would eradicate worms. Although significant progress has been made on software development, testing, and verification, empirical evidence [CER05] suggests that we are still far from producing defect-free software. Figure 1.1 shows that for the last five years exploitable software defects (also referred to as vulnerabilities) have been discovered at a rate of several thousands per year.

Another avenue to solve the worm problem is containment. Containment systems accept that software has defects that can be exploited by worms, and they strive to contain a worm epidemic to a small fraction of the vulnerable

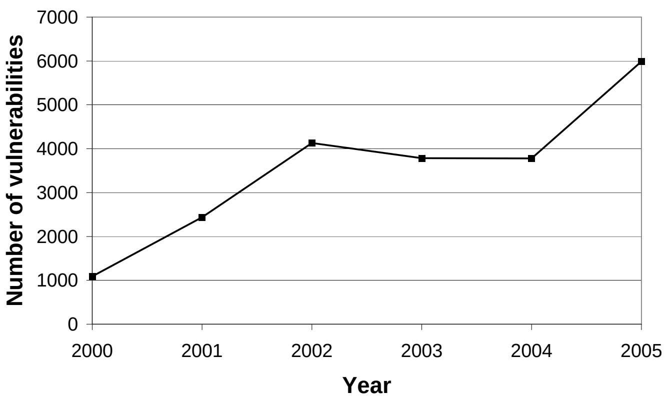


Figure 1.1: Number of new vulnerabilities reported to CERT from the year 2000 to the year 2005.

machines. The main challenge in designing containment systems is that they need to be completely automatic, because worms can spread far faster than humans can respond [MSVS03; SPW02]. Recent work on automatic containment [KK04; SEVS04; KC03; WSP04] has explored *network-level* approaches. These rely on heuristics to analyze network traffic and derive a packet classifier that blocks or rate-limits forwarding of worm packets. It is hard to provide guarantees on the rate of false positives and false negatives with these approaches because there is no information about the software vulnerabilities exploited by worms at the network level. False negatives allow worms to escape containment, while false positives may cause network outages by blocking normal traffic. We believe that automatic containment systems will not be widely deployed unless they have a negligible false positive rate.

This thesis proposes Vigilante, a new worm containment architecture that addresses these limitations by using an *end-to-end* approach. Vigilante's design is fully automatic, has no false positives, provides broad coverage of worm attacks, and allows vulnerable software services to continue to run efficiently while being attacked. Rather than relying only on network-level information, Vigilante uses information available at *end-hosts*. Since hosts run the software that is vulnerable to infection by worms, they can analyze the software to gather information that is not available to network-level approaches. Vigilante leverages this information to contain worms that escape network-level detection and to eliminate false positives.

Figure 1.2 illustrates automatic worm containment with Vigilante. In Vigilante, hosts detect worms by instrumenting network-facing services to analyze infection attempts. Vigilante introduces dynamic data-flow analysis: a host-based algorithm that can detect worms that exploit unknown vulnerabilities with broad coverage. Dynamic data-flow analysis tracks the flow of data from network messages inside the vulnerable program, and disallows unsafe uses of the data. The algorithm detects the three most common infection techniques used by worms: code injection, edge injection and data injection on unmodified binaries. We also show how to integrate other detection mechanisms into the Vigilante architecture. Using host-based infection detectors enables Vigilante to detect worms that have normal network traffic patterns, since at some point they still need to infect their victims.

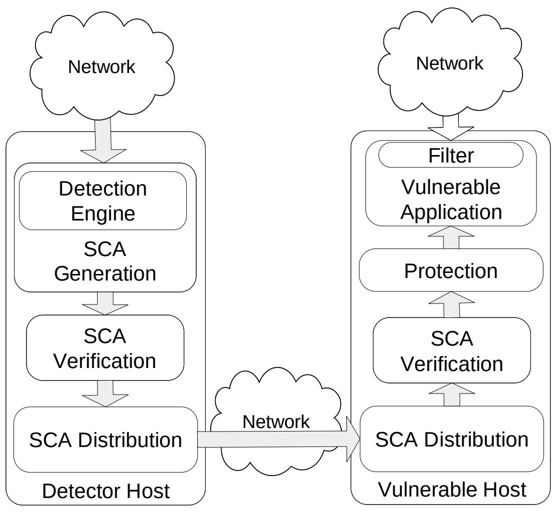


Figure 1.2: Automatic worm containment in Vigilante.

Vigilante relies on collaborative worm detection at end hosts, but does not require hosts to trust each other. Upon detection, hosts generate self-certifying alerts (SCAs). An SCA is a machine-verifiable proof of vulnerability: it proves the existence of a vulnerability in a service and can be inexpensively verified. By verifying an SCA, a host can determine with certainty that a software service is vulnerable; the verification procedure has no false positives. SCAs enable cooperative worm detection with many detectors distributed all over the network, thereby making it hard for the worm to avoid detectors or to disable them with denial-of-service attacks. Additionally, cooperation allows hosts to run expensive detection engines with high accuracy, because it spreads detection load. For example, a host that does not run a database server can run a version of the server instrumented to detect infection attempts in a virtual machine. This instrumented version is a honeypot; it should not normally receive traffic. Therefore, the host will incur little overhead for running the detection engine, whereas a production database server could incur an unacceptable overhead.

SCAs also provide a common language to describe vulnerabilities and a common verification mechanism, which can be reused by many different detection engines to keep the trusted computing base small. SCAs could be verified using the detection engine that generated them but this would require all vulnerable hosts to run and trust the code of all detection engines. SCAs make it possible to increase aggregate detection coverage by running many distinct detection engines and by deploying new engines quickly.

In Vigilante, detectors distribute SCAs to other hosts using an overlay network that propagates alerts rapidly and resiliently. Before a host distributes an SCA or after it receives an SCA from another host, it verifies the SCA by reproducing the infection process described in the SCA in a sandbox. If verification is successful, the host initiates the process of protecting the vulnerable service.

Alerted hosts protect themselves by generating filters that block worm messages before they are delivered to a vulnerable service. These filters are generated automatically using dynamic data and control flow analysis of the execution path followed by a worm when exploiting the vulnerability described in an SCA. This procedure, called *vulnerability condition slicing*, identifies a set of instructions in the program that define control-flow conditions that lead to successful attacks. Filters block messages that satisfy these conditions. Each vulnerable host runs this procedure locally and installs the filter to protect itself from the worm.

Since this procedure runs locally, hosts don't need to trust external entities to produce correct filters. Furthermore, analyzing the vulnerable code, instead of the worm attack messages, allows hosts to generalize the attack beyond what they observed. It also avoids interference by the worm, since the vulnerable code is not controlled by the worm, while the attack messages are.

The filters generated by *vulnerability condition slicing* introduce low overhead, have no false positives, and block all worm mutations that follow the same execution path to gain control.

1.1 Contributions

This thesis presents Vigilante, a new system to contain worm epidemics automatically with an end-to-end architecture. An end-to-end architecture can use information about the vulnerable software services running at end-hosts, which is not available to network-level approaches. This information can be leveraged to detect worms even when they exhibit normal traffic patterns, to verify unequivocally that a host is running vulnerable software and to guide the generation of filters that block mutations of a worm attack. An end-to-end architecture ensures that the containment system will react only when real worm outbreaks happen, and it can contain worms that escape network-level containment.

The contributions of this thesis are:

- an *end-to-end architecture* for automatic worm containment based on high-coverage host-based detectors, security alerts that can be verified, an overlay network for fast and resilient dissemination of alerts, and automatic generation of filters to prevent infection at end-hosts,
- a *dynamic data-flow analysis algorithm* that provides automatic high-coverage detection of worm infection attempts for unknown vulnerabilities,
- the concept of *self-certifying alerts* (SCAs) and mechanisms to generate, verify, and distribute SCAs automatically,
- a *vulnerability condition slicing algorithm* to automatically generate host-based filters that block worm infections, and
- experimental evaluation of the architecture and algorithms through measurements of a prototype implementation and large scale simulation.

To validate the system, we implemented the algorithms on Intel IA-32 machines running the Windows operating system. We tested Vigilante with a broad range of synthetic attacks and also with three infamous real worms: Slammer, CodeRed, and Blaster. We also simulated the overall behaviour of the system when deployed on the Internet. We parameterized the simulations with measurements from our implementation and with data from Internet measurements of notorious worm outbreaks. Our results show that Vigilante can contain fast spreading worms that exploit unknown vulnerabilities, even when only a small fraction of the vulnerable hosts can detect the attack. The results also show a negligible performance impact on the vulnerable software services protected by Vigilante.

Furthermore, Vigilante does not require any changes to hardware, compilers, operating systems or the source code of vulnerable programs; therefore, it can be used to protect software as it exists today in binary form.

1.2 Organization

The rest of this dissertation is organized as follows. Chapter 2 describes in detail how worms infect their victims, how they spread on the Internet and how they try to avoid being detected. In Chapter 2, we also discuss how the characteristics of worm attacks have guided our design of Vigilante. Chapter 3 describes the design and implementation of dynamic data-flow analysis, and discusses the importance of using a diverse set of detection mechanisms. Chapter 4 introduces the concept of SCA and describes procedures to verify, generate, and distribute SCAs. The design and implementation of the automatic filter generation mechanism is presented in Chapter 5. Chapter 6 presents our experimental results. Chapter 7 describes related work. We conclude in Chapter 8 and discuss some directions for future work.

Chapter 2

Worm Attacks

Worms[1] are computer programs that self-replicate without requiring any human intervention, by sending copies of their code in network packets and ensuring the code is executed by the computers that receive it. When computers become infected, they spread further copies of the worm and possibly perform other malicious activities.

The first experiments with programs similar to modern worms were reported in 1982 [SH82]. However, worms did not become a major security threat until the advent of the Internet: by connecting most of the world's computers, the Internet enabled global worm epidemics. The first recorded Internet worm outbreak happened in 1988 [Spa89; ER89]; since then, several major incidents have been reported [MSB02; BCJ$^+$05; MPS$^+$03; SM04].

In this chapter, we analyze how worms infect remote computers, how they spread on the Internet, and how they try to avoid being detected. We also discuss how the techniques used by worms guided our design of Vigilante. The many ways in which worms can change their behaviour in response to containment systems led us to design Vigilante in a *vulnerability-centric* way: all the mechanisms introduced by Vigilante are based on analyzing the vulnerable code infected by worms. This approach is more effective than analyzing the worm's code or the worm's behaviour, because the vulnerable code is not under the control of the worm.

[1]The use of "worm" with this meaning derives from the "tapeworm" programs in John Brunner's 1975 novel *The Shockwave Rider*.

2.1 Infection

Remotely infecting a computer requires coercing the computer into running the worm code. To achieve this, worms exploit low-level software defects, also known as vulnerabilities. Vulnerabilities are common in current software, because it is large, complex, and mostly written in unsafe programming languages. Several different classes of vulnerabilities have been discovered over the years. Currently, buffer overflows [One96], arithmetic overflows [ble02], memory management errors [jp03], and incorrect handling of format strings [gr02] are among the most common types of vulnerabilities exploitable by worms.

While we should expect new types of vulnerabilities to be discovered in the future, the mechanisms used by worms to gain control of a program's execution should change less frequently. Currently, worms gain control of the execution of a remote program using one of three mechanisms: injecting new code into the program, injecting new control-flow edges into the program (e.g. forcing the program to call functions that shouldn't be called), and corrupting data used by the program. Vigilante was designed to detect all three kinds of infection.

The next sections discuss these three infection mechanisms in detail. To facilitate the discussion, we use a program with simple stack-based buffer overflow vulnerability [One96], shown in Figure 2.1, but it is important to note that all the other types of vulnerabilities enable the same types of successful infection. The program in Figure 2.1 processes a message received from the network. The *ProcessRequest* function checks if the first byte in the message is within an allowed range, and then copies the two subsequent fields in the message to a stack-based buffer called *request* and to a buffer supplied in the parameter *userid*, respectively. The code assumes fields in the message are separated by the newline character. The defect in the program is that it does not check that the first field of the message will fit the in the *request* stack-based buffer. Consequently, the worm can send a message with a large first field and overwrite the stack frame [ASU86]. This defect can be exploited to infect the program in many ways.

2.1.1 Code injection

The simplest form of infection involves injecting new code into a running process and coercing the process into executing the new code. To use this type of attack on the program in Figure 2.1, the worm could craft a packet including its code at the end of the message and using a first field large enough to overwrite the return address on the stack frame. Inside the first field, at the position that would overwrite the return address, the worm would supply the address of its code in the virtual address space of the program under attack (the code would be there as part of the message just received). This would ensure that, upon executing the *ret* instruction at the end of the function, the process would start to run worm code.

The details of the attack can be understood by analyzing the vulnerable program in assembly language, as shown in Figure 2.2 (Appendix A describes the instructions used in the example). When the *ProcessRequest* function starts to execute, the *esp* register points to the return address saved by the *call* instruction that transferred control to the function. The function starts by saving the *ebp* register on the stack, decrementing *esp* by 4 in the process (the stack grows towards lower addresses). Instruction 3 moves the first byte of the message into the *al* register (the first parameter for the function is passed in the *ebx* register). The function then executes the range check on the first byte of the message. Instruction 7 subtracts 8 from *esp*, thus allocating 8 bytes on the stack, to hold the *request variable*. Therefore the return address is stored at a 12 byte offset from start of *request*. This means that the worm should place the value to be used as return address at offset 13 in the attack message (since the first byte is not copied). Instruction 16 makes *eax* point to the start of the *request* buffer. The function then enters a loop (lines 21 to 26) that copies the first field of the message and eventually overwrites the stored return address. To decide which value to supply as return address, the worm only needs to know the virtual address range where the network message is stored and use a value that points to the start of the worm code within that range[1].

[1]If the message is not stored at a predictable address, the worm can find code sequences that transfer control to the attack payload elsewhere in memory [ds99].

```
void ProcessRequest(char *message, char *user_id)
{
    char request[8];
    char message_id = *message - ID_BASE;

    if(message_id > LOW_ID && message_id < HIGH_ID)
    {
        int len = CopyField(request,message + 1);
        CopyField(user_id,message + len + 2);
        ExecuteRequest(request,user_id);
    }
    system(log_activity_command);
}

int CopyField(char *destination, char *source)
{
    int len = 0;
    while(*source != '\n')
    {
        len++;
        *destination++ = *source++;
    }
    *destination = '\0';
    return len;
}
```

Figure 2.1: Vulnerable code in the C++ programming language. The code has a buffer overflow vulnerability enabling code injection, edge injection, and data injection attacks.

```
 1: push    ebp                    ;on entry, ebx points to the message parameter
 2: mov     ebp,esp
 3: mov     al,byte ptr [ebx]      ;move first byte of message into al
 4: mov     ecx,dword ptr [ebp+8]
 5: sub     al,10h
 6: sub     al,31h
 7: sub     esp,8                  ;allocate stack space for request buffer
 8: cmp     al,0Eh                 ;perform range check on first byte
 9: ja      45
10: mov     dl,byte ptr [ebx+1]    ;move second byte of message into dl
11: push    esi
12: push    edi
13: lea     edi,[ebx+1]            ;move address of second byte into edi
14: xor     esi,esi
15: cmp     dl,0Ah
16: lea     eax,[ebp-8]            ;move address of request buffer into eax
17: je      28
18: mov     ecx,eax
19: sub     edi,ecx
20: lea     esp,[esp+0h]
21: mov     byte ptr [eax],dl      ;loop to copy the first
22: mov     dl,byte ptr [edi+eax+1] ;field of the message
23: add     eax,1                  ;into the request buffer,
24: add     esi,1                  ;while searching for
25: cmp     dl,0Ah                 ;the character 0A.
26: jne     21
27: mov     ecx,dword ptr [ebp+8]  ;move userid parameter into ecx
28: lea     esi,[esi+ebx+2]
29: mov     byte ptr [eax],0
30: mov     al,byte ptr [esi]      ;move first byte of second field into al
31: cmp     al,0Ah
32: mov     edx,ecx                ;move userid parameter into edx
33: je      40
34: sub     esi,ecx
35: mov     byte ptr [edx],al      ;loop to copy the second
36: mov     al,byte ptr [esi+edx+1] ;field of the message into
37: add     edx,1                  ;the userid parameter, while
38: cmp     al,0Ah                 ;searching for the character 0A.
39: jne     35
40: lea     eax,[ebp-8]
41: mov     byte ptr [edx],0
42: call    ExecuteRequest         ;call ExecuteRequest(request,user_id)
43: pop     edi
44: pop     esi
45: push    403018h                ;push address of log_activity_command
46: call    system                 ;call system(log_activity_command)
47: add     esp,4
48: mov     esp,ebp
49: pop     ebp
50: ret                            ;load value pointed to by esp into eip
```

Figure 2.2: Vulnerable program in IA-32 assembly language (compiled from the source code in Figure 2.1). The code is vulnerable to code injection, edge injection, and data injection attacks.

2.1.2 Edge injection

Infecting a remote computer does not require directly injecting new code into a running process. Another way to carry out infection, is to inject a new control-flow edge into the vulnerable program by forcing a control-flow transition that should not happen [Ner01]. To use this type of attack on the program in Figure 2.1, the worm could again craft a message including a first field large enough to overwrite the return address on the stack frame. This would allow the worm to supply as a return address, the address of a function already loaded by the program. For instance the attacker could supply the address of the *system* function from the C runtime library, and an appropriate argument to that function. This would allow the worm to run arbitrary programs. It could, for instance, use a file transfer program to download its code and subsequently run it. This attack can evade algorithms that only detect code injection, because no new code is loaded by the process running the vulnerable program.

The detailed steps of the attack are similar to the ones described in the previous section, with the added complication that the worm needs to fabricate a correct stack frame for the *system* function. This can easily be accomplished by noting that this function takes a single parameter: a pointer to a string. Thus, the code for the function expects to find this pointer at the address 4 bytes above the value of the *esp* register when the function is invoked (at this point *esp* points to the return address and the parameter is above that). Consequently, besides supplying the address of the *system* function as in the previous section, the worm will insert in the message a string with the name of the program it wishes to run, and 8 bytes after the start of the address of *system* (i.e. at offset 21 in the attack message) it will supply the address where the string will be stored in the virtual address space of the target program. The worm can easily extend this technique to fabricate several stack frames and force the program to issue a series of function calls [Ner01].

2.1.3 Data injection

Finally, infecting a remote computer does not even require forcing any control-flow error in a running process: attacks can succeed just by corrupting data. One general form of this type of attack involves corrupting the arguments of functions called by the program. By changing the values of the arguments, the worm changes the behaviour of the program, without injecting any code or forcing any control-flow transfers.

Using again the example in Figure 2.1, we can see that, after processing the message, the function *ProcessRequest* calls *system* to run an external program that maintains an activity log for the program. The call to *system* takes as parameter a pointer (*log-activity-command*) to a string with the appropriate logging command. The worm can successfully attack the program by corrupting this string, thus forcing the program to run other commands (e.g. commands that download the worm code). Corrupting this string is a slightly more elaborate process than corrupting the function's return address, because neither the string nor the pointer to the string are stored in the stack frame for the function (the region that can easily be overwritten by overflowing the *request* buffer). However, the worm can still manipulate the code in the function to do the appropriate overwrite. It notes that the code copies the second field in the message to the *userid* parameter. This parameter is in the function's stack frame and can be easily overwritten. Therefore all the worm needs to do is to overwrite *userid* to make it point to the *log-activity-command* string and supply, as the second field in the attack message, a string with the command it wishes to run.

The detailed steps of the attack can again be understood by analyzing the code in Figure 2.2. The code reveals that the *userid* argument is passed to the function on the stack, immediately above the return address. To see this, note that instruction 27 loads the *userid* pointer into the *ecx* register, and instructions 35 to 39 copy the second field of the message into *userid*. As in the attacks above, the worm can supply a large first field in the attack message, overflowing the *request* buffer. This allows the worm to supply a value that will overwrite *userid* at offset 17 in the attack message. Examining the call to *system* at lines 45 and 46, we can see that the *log-activity-command* is stored at address 0x00403018. Therefore, the worm supplies this value at offset 17 in the attack message and whatever command it wants to run as a string in the second field in the message. Thus, the loop at lines 35 to 39, which should copy the second field in the message into *userid* is in fact copying the second field into the *log-activity-command* string. The worm command is executed when the program calls *system* on line 46.

2.2 Spreading

After infecting a computer, worms typically use it to infect other computers, giving rise to a propagation process which has many similarities with the spread of human diseases. Empirical evidence [MPS+03; SPW02] shows that the spread of real worms, such as Code Red and Slammer, can described using the epidemic model for infectious diseases described in [Het00]. Assuming a population of S susceptible hosts and an average infection rate of β, and using I_t as the total number of infected hosts at time t, the worm infection is modelled by the following equation:

$$\frac{dI_t}{dt} = \beta\ I_t(1 - \frac{I_t}{S}) \tag{2.1}$$

Figure 2.3 plots the propagation of a Slammer-like worm, as predicted by the model. It shows that initially the number of infected hosts grows exponentially until a majority of hosts are infected. The model matches accurately the initial stages of the Code Red and Slammer outbreaks [MPS+03], but later stages tend to deviate from the model due to network congestion effects or human intervention. For the purpose of designing a containment system such as Vigilante, the model is appropriate because the containment system must react during the initial stages of the outbreak, to have any chance of saving a significant fraction of the vulnerable population. Vigilante was designed to fulfil the requirements for reaction time identified by Moore et al. [MSVS03]: it typically activates filtering mechanisms automatically within seconds of the start of an epidemic.

The speed of propagation of worms depends on how fast infected computers can find new victims to infect, and worms can find new victims in many ways [SPW02; WPSC03]. Scanning worms send attack messages to Internet Protocol (IP) addresses that they generate locally, trying to scan the complete IP address space. The address space may be scanned randomly, linearly or trying

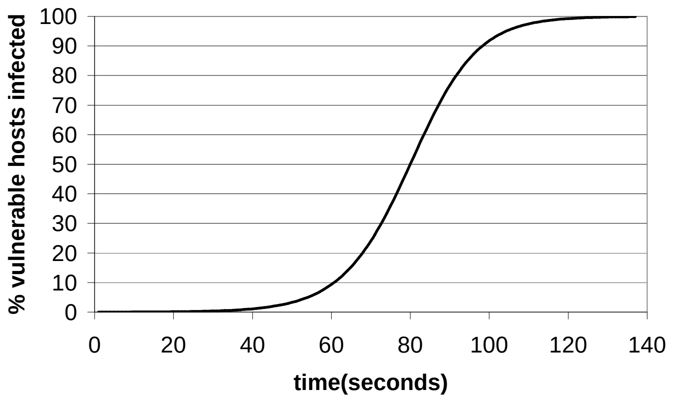


Figure 2.3: Example of propagation of a Slammer-like worm.

to give preference to address regions that are more likely to be populated with victims [SPW02].

Topological worms find new victims by finding their addresses in the infected computers. Thus, they spread along a topology maintained by the computers they infect. There are many examples of such topologies, for example, peer-to-peer networks [RD01; SMK+01; RFH+01] form well-connected topologies that can be exploited in this way. Game server farms, where many computers connect to a few servers, can also facilitate this type of spreading.

Hitlist worms compute a list of victims before starting the attack, thus avoiding the need to discover victims during the attack. Such lists can be obtained with different online or offline means; Internet search engines and configuration files are good sources for this kind of information. During the attack, these worms possibly partition the list of victims among all the infected machines, to speed up the attack.

All of these strategies for finding new victims have been observed in one form or another on the Internet. Worms can also combine several strategies. For instance the Witty [SM04] worm used a hit list to ramp up the initial phase of its spread, and then switched to scanning the Internet for subsequent victims.

Vigilante is agnostic to the propagation strategy used by worms. Whatever the mechanism used by a worm to find new victims, it needs to infect them by exploiting a software vulnerability. Vigilante detects the worm during the infection process.

2.3 Hiding

Worms can use several techniques to disguise their spread on the Internet. In this section, we focus on three evasion techniques that guided our design of Vigilante: traffic shaping, polymorphism, and fingerprinting detectors.

2.3.1 Traffic shaping

Worms usually have complete control over the network traffic generated by the computers they infect. This means they can blend attack traffic with normal traffic, making it difficult to detect them by analyzing traffic patterns. For instance, some detection and mitigation systems are based on the observation that scanning worms are likely to target many invalid addresses [Pax99; WSP04]. Some of these systems use a limit on the ratio of failed to successful connections: traffic from computers that exceed this limit is blocked. These systems can easily be evaded if the worm establishes a successful connection to another worm instance for each address that it scans. Other systems assume that worms must initiate connections to other computers at a high rate [Roe99; HDK+90; Wil02], to propagate rapidly. These systems detect worm traffic by monitoring the rate at which unique destination addresses are contacted, and block the sender if the rate exceeds some limit. These systems can be evaded if worms initiate connections just below the rate limit, in areas of the network where the limits are enforced. Vigilante cannot be evaded with traffic shaping attacks, because it uses host-based detectors.

Traffic shaping can also be used to mount denial of service attacks on systems that analyze network traffic [WSP04; PDL+06]. Worms can, for instance, generate suspicious traffic patterns using fake IP source addresses, in order to block traffic from legitimate machines. This type of maliciously induced false positives is a serious concern for the deployment of containment systems based exclusively on analyzing network traffic.

2.3.2 Polymorphism

Another powerful technique that worms can use to hide themselves is polymorphism[1]. Polymorphic worms constantly change the content of their attack messages using techniques such as encryption and code obfuscation [SF01]. For instance, if the worm attack messages contain code, the worm can replace a sequence of code instructions with another completely different sequence of instructions that achieves the same purpose (i.e. a semantically equivalent sequence of instructions). Figure 2.4 shows an example of this type of code mutation. Worm writers can use readily available tools to create polymorphic worms [Z0m00; K201; DUYU03; Ban05].

Polymorphism creates problems both for detection and for protection. From the detection point of view, systems that try to detect worms by identifying common byte strings in suspicious network traffic [KK04; SEVS04], will have difficulty detecting polymorphic worms, since they may have little invariant content across different messages. For instance, a polymorphic worm exploiting the same vulnerability as Slammer [MPS$^+$03] could generate attack messages that differ in all bytes, except one. On the other hand, if such systems are configured to detect very short repeated byte strings, they are likely to generate false positives. Vigilante avoids these problems with detection, by using host-based detectors that do not rely on analyzing network traffic.

From the protection point of view, polymorphic worms make it difficult to block attacks using byte string signatures [KC03; KK04; SEVS04], because such signatures will likely be either too long or too short. Signatures that are too long cannot match worm traffic mutations and signatures that are too short will block normal traffic.

Vigilante addresses this problem by generating protection filters that are programs, not byte strings. Vigilante's filters can compute complex conditions on the attack messages. Vigilante generates these filters by analyzing the vulnerable programs, not network traffic. The vulnerable code provides information to

[1]We use the term polymorphic to refer to polymorphic, oligomorphic, and metamorphic worms as described in [SF01].

```
Original code:
mov ebx,eax
inc ebx
sbb eax,eax
dec ebx
add eax,0x3
dec eax

Mutation:
push ecx
push eax
xor ecx,ecx
inc ecx
pop ebx
mov eax,ecx
pop ecx
inc eax


Byte representation for original code:
8B D8 43 1B C0 4B 83 C0 03 48

Byte representation for mutated code:
51 50 33 C9 41 5B 8B C1 59 40
```

Figure 2.4: Example of polymorphic worm code. Both the original code and the mutation assign *eax* to *ebx* and assign 2 to *eax*, but they use different instructions to do so. This results in different byte representations in memory and in attack messages.

identify why the attack succeeded; this information cannot be found in samples of attack messages.

2.3.3 Fingerprinting

Another technique that worms can use to avoid being detected is to try to identify if they are interacting with a detector, before fully revealing their attack. We refer to this type of activity as *fingerprinting* the detector [HR05; BFV05; SII05].

Worms can try to fingerprint detectors remotely, i.e. try to infer if a remote machine is a detector from its responses to network messages. Some honeypot systems [Pro04] mimic responses to commonly used protocols, without fully implementing the corresponding network services. These systems are more vulnerable to fingerprinting, because they cannot generate the full spectrum of responses expected from a real network service. Detectors in Vigilante run full operating system and applications software, thereby minimizing their exposure to this type of attack.

Another form of remote fingerprinting involves analyzing the timing of message exchanges. For instance, if responses take more time than normal, the remote system may be a detector. It is unclear if this type of attack can be mounted across wide area networks, where the CPU speeds, load, and latency to remote machines is unknown. Vigilante mitigates this type of attack by using different types of host-based detectors spread over the network, thus making their timing behaviour harder to predict.

Worms can also fingerprint detectors locally, after they start to run on an infected machine. For instance, if a detector is trying to identify a certain type of behaviour (e.g. a malicious pattern of usage of operating system services), the worm can first check if it is running inside the detection environment and, if so, take evasive actions. This type of fingerprinting has been observed on the Internet. Holz et al. [HR05] report that *Agobot* uses an I/O backdoor [Kat06] to detect if it is running inside a virtual machine. Agobot also detects the presence of debuggers and breakpoints. Vigilante avoids local fingerprinting by using detectors that do not allow the worm to run any instructions. Since the worm code does not run, it can't take evasive actions. The next chapter explains how to build such a detector.

Chapter 3

Detection

The first step towards containing the outbreak of an unknown worm is to detect it. Vigilante detects worms by analyzing the execution of vulnerable programs. To be effective, worm detectors need to have high coverage and generate few false positives. This chapter introduces dynamic data-flow analysis, a new host-based detection algorithm that achieves these goals. We also discuss the importance of using a diverse set of detection mechanisms in Vigilante.

3.1 Dynamic Data-flow Analysis

Remotely exploiting software defects requires sending messages that enable the attacker to gain control of the target computers. Therefore, all remote attacks can be linked to errors that occur while processing messages received from the network. The dynamic data-flow analysis detection algorithm is based on the idea of dynamically tracking the flow of data received from the network and disallowing unsafe uses of that data.

3.1.1 Algorithm

The dynamic data-flow analysis algorithm, shown in pseudo-code in Figure 3.1, consists of two parts. The first part tracks data received from the network. Whenever a network input operation completes, the memory locations where the data is written are marked *dirty*. Then, the algorithm tracks all movements of that data. Whenever the processor executes an instruction that moves data from a source to a destination, the destination becomes dirty if the source is dirty or it becomes clean otherwise. Sources and destinations can be memory locations or processor registers. At all times, the algorithm keeps track of the location of all copies of data received from the network.

```
OnReceivedNetworkMessage($address$, $size$)
    for $i \leftarrow 0$ to $size - 1$
        do
            SetDirty($address + i$)

OnMoveDataInstruction($destination$, $source$)
    if IsDirty($source$)
        then SetDirty($destination$)
        else ClearDirty($destination$)

OnIndirectControlFlowTransition($address$)
    ▷ halt if the program counter is loaded from a dirty $address$ or is directed to dirty memory
    if IsDirty($address$)
        then GenerateSecurityTrap
    if IsDirty(ValueAt($address$))
        then GenerateSecurityTrap

OnDirectControlFlowTransition($address$)
    ▷ halt if the program counter is directed to a dirty $address$
    if IsDirty($address$)
        then GenerateSecurityTrap

OnNextInstruction($address$)
    ▷ halt if execution falls-through to a dirty $address$
    if IsDirty($address$)
        then GenerateSecurityTrap

OnSecuritySensitiveFunctionCall($argument$)
    if IsDirty($argument$)
        then GenerateSecurityTrap
```

Figure 3.1: Dynamic data-flow analysis algorithm.

The second part of the algorithm generates a security trap when dirty data is used in an unsafe way. To decide which uses of dirty data are unsafe, we need to consider the ways in which worms can infect a running process. As discussed in Chapter 2, worms can infect a process using three types of attacks: injecting new code into the process, injecting a new control-flow edge into the process (i.e. forcing the process to make an unwanted control-flow transition), and injecting data used in security sensitive operations. To prevent each of these types of infection, dynamic data-flow analysis generates a security trap on each of the following situations:

i. execution of dirty data,

ii. loading of dirty data into the program counter, and

iii. passing of dirty data in arguments of security sensitive functions.

Preventing execution of dirty data is important, because the data came from network messages and therefore corresponds to code injected by the worm. Preventing loading of dirty data into the program counter is important, because by supplying data used in this way the worm can force the program to make arbitrary control-flow transitions. Finally, passing data to arguments of security sensitive functions is a common form achieving infection by only injecting data; therefore, it is also important to prevent it.

To be able to generate security traps on the first and second conditions above, the algorithm dynamically analyzes the state of memory and processor registers at every control-flow transition in the program. If the execution is being directed to memory region that contains dirty data or if the data loaded into the program counter is dirty, a security trap is generated. To enforce the third condition, whenever security sensitive functions are called, their arguments are checked for dirtiness. For instance, when operating system functions that create new processes are called, the argument that specifies the program to run is checked for dirtiness, because controlling this argument would allow the worm to launch arbitrary programs.

We will use the vulnerable code in Figure 3.2 to illustrate how the dynamic data-flow analysis algorithm can detect an edge injection attack (the mechanics of attacks on this code were described in Chapter 2). When the code starts to execute, the *esp* register points to the return address saved by the *call* instruction

```
 1: push     ebp                    ;on entry, ebx points to the message parameter
                                    ;esp points to eip saved on the stack
                                    ;the memory containing the message is marked dirty
 2: mov      ebp,esp
 3: mov      al,byte ptr [ebx]      ;move first byte of message into al
                                    ;mark al as dirty
 4: mov      ecx,dword ptr [ebp+8]
 5: sub      al,10h
 6: sub      al,31h
 7: sub      esp,8                  ;allocate stack space for request buffer
 8: cmp      al,0Eh                 ;perform range check on first byte
 9: ja       45
10: mov      dl,byte ptr [ebx+1]    ;move second byte of message into dl
                                    ;mark dl as dirty
11: push     esi
12: push     edi
13: lea      edi,[ebx+1]            ;move address of second byte into edi
14: xor      esi,esi
15: cmp      dl,0Ah
16: lea      eax,[ebp-8]            ;move address of request buffer into eax
17: je       28
18: mov      ecx,eax
19: sub      edi,ecx
20: lea      esp,[esp+0h]

21: mov      byte ptr [eax],dl      ;copy next byte into request buffer
                                    ;mark address pointed to by eax as dirty
22: mov      dl,byte ptr [edi+eax+1] ;move next byte of message into dl
                                    ;mark dl as dirty
23: add      eax,1
24: add      esi,1
25: cmp      dl,0Ah                 ;if not found 0A, continue to next byte.
26: jne      21

...                                 ;irrelevant instructions omitted

48: mov      esp,ebp
49: pop      ebp
50: ret                             ;load value pointed to by esp into eip
                                    ;generate a security trap, because
                                    ;esp points to dirty memory
```

Figure 3.2: Example of detection with vulnerable program in IA-32 assembly language (compiled from the source code in Figure 2.1).

that transferred control to the function, and the *ebx* register holds the *message* parameter. The parameter points to a message just received from the network. When the message was received, the memory pointed to by *ebx* was marked dirty. After executing some instructions irrelevant for the attack, the program reaches instruction 7 which subtracts 8 from *esp*, thus allocating 8 bytes on the stack to hold the *request variable*. After running the range check on the first byte of the message, on line 8, the program loads the second byte of the message into the *dl* register, on line 10. At this point *dl* is marked dirty, because the memory at *ebx+1* is dirty. Instruction 16 makes *eax* point to the start of the *request* buffer. The function then enters a loop, on lines 21 to 26, that copies the first field of the message. When instruction 21 executes, the memory location pointed to by *eax* is marked dirty, because *dl* is dirty. Instruction 22, loads the next byte of the message into *dl*, which remains dirty. The byte is then compared with the newline character (*0x0A*), and the loop continues if the newline was not reached. The loop eventually overwrites the stored return address, and the memory location where the return address is stored is marked dirty in the process. Figure 3.3 shows the state of memory just before and immediately after the vulnerable code is executed. After executing some more instructions irrelevant for the attack, the code reaches the *ret* instruction at line 50. At this point, the algorithm generates a security trap, because the *esp* register points to a dirty memory location — the location where the return address was originally stored. Thus, dynamic data-flow analysis detects the worm before it can inject an arbitrary control-flow edge into the program by supplying a value to be used as return address.

The dynamic data-flow analysis algorithm has several important properties. First, it has broad coverage: it detects the three kinds of infection mechanisms most used by worms. It detects overwrites of control data structures with data received from the network, and it prevents execution of data received from the network. Furthermore, it detects attacks that do not cause control-flow errors in the program. As we have shown in section 2.1, the same vulnerability can often be exploited to infect a program with these three different techniques; hence it is important to detect all of them.

Second, dynamic data-flow analysis is independent of vulnerabilities and attack targets. Since the algorithm does not require any information about vulner-

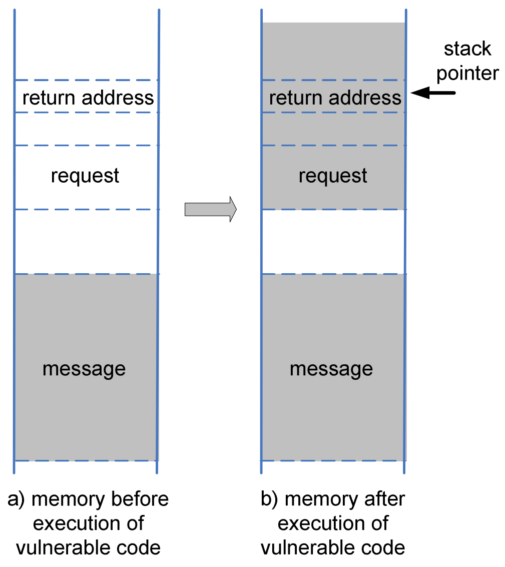

return address
request
message
return address
request
message
stack pointer
a) memory before execution of vulnerable code
b) memory after execution of vulnerable code

Figure 3.3: Example of worm detection with dynamic data-flow analysis. The figure shows the memory when (a) a message is received and the vulnerable code is about to execute, and (b) after the vulnerable code executes and overwrites the return address in the stack. Greyed areas indicate dirty memory regions.

abilities or targets of attacks inside programs, it will remain useful if new types of vulnerabilities or attack targets are identified in programs in the future. In contrast, previous techniques that protect specific targets in programs have been shown to be easy to bypass [WK03]. For instance, mechanisms that protect return addresses on the stack [CPM+98] can be bypassed by overwriting function pointers.

Third, dynamic data-flow analysis works on unmodified programs in binary form. The algorithm inspects execution at the processor instruction level, consequently it does not require source code or any form of cooperation from the entity producing the program under analysis. Thus, it can be used to detect infection of the normal binaries currently deployed. Furthermore, it works even with self-modifying and dynamically generated code.

Finally, the dynamic data-flow analysis prevents evasive action by the worm. The algorithm detects the worm infection attempt before the worm executes any instructions. This is a key property, because it prevents the worm from checking that it is running in a detection environment and using evasion techniques. For instance, if the worm was allowed to execute any instructions it could time its own execution to try to distinguish a normal execution from an execution inside the detection environment, or use one of other fingerprinting techniques discussed in Section 2.3.3.

Dynamic data-flow analysis has several limitations. One of the limitations is that it may generate false positives. This happens because programs may perform safety checks on data received from the network before using that data in ways that would be unsafe if the checks were not performed. Since dynamic data-flow analysis is unaware of the checks performed by the programs, it still generates a security trap on the potentially unsafe uses of the data. This may happen, for instance, when a program loads a value received from the network into the program counter after checking that the value is within a safe range (e.g. checking that the value is the address of a function in the program). Our experiments in Chapter 6 indicate that such cases are rare. Another example is an application (e.g. the web browser) for which downloading and executing code is a normal activity. Such applications already have mechanisms to control the execution of the downloaded code (e.g. requiring it to be signed by a trusted entity). These mechanisms would need to be integrated with dynamic data-flow analysis to enable the applications to run (e.g. by explicitly informing the dynamic data-flow analysis algorithm that a piece of downloaded code is granted execution privileges). This type of integration is a simple operation and it needs to be done only once for each specific code downloading mechanism. It is also important to note that the vast majority of programs are not designed to download code at runtime, and thus they are not affected by this restriction.

Even if they are rare, false positives are a serious concern, because organizations will understandably avoid deploying automatic worm containment systems, if those systems may generate security alerts and block traffic even when there is no worm outbreak. To address this problem, we describe in the next chapter a verification mechanism to discard any false positives generated by detectors in the Vigilante containment system.

The other limitation of dynamic data-flow analysis is that it may have false negatives, i.e. there are several attacks that it cannot detect. Dynamic data-flow analysis cannot detect attacks that exploit high-level defects in programs such as explicit backdoors in programs. Backdoors can exist either due to malicious intent of the developers who wrote the code, or they may simply be due to unintentional development mistakes. Dynamic data-flow analysis also cannot detect software configuration errors such as weak passwords. Accessing a computer with a guessed or stolen password is indistinguishable from a legitimate access.

Finally, dynamic data-flow analysis will not detect attacks that overwrite security sensitive information with values controlled by the worm, but not directly copied or derived from the attack messages. Two important cases where this may happen are when network data is combined with other data through arithmetic and logic operations, and when network data is used to control the addresses of load and store instructions in the program. These types of false negatives can be addressed by extending the algorithm to propagate dirtiness to the destination operands of arithmetic and logic instructions and to the destination operands of loads and stores. The extended algorithm would provide increased coverage, but would also increase the number of false positives.

3.1.2 Implementation

The dynamic data-flow analysis algorithm can be implemented in several ways. It can be implemented in hardware by changing the processor's data movement instructions to propagate dirtiness and augmenting the instructions that change control-flow with checks to avoid loading dirty data into the program counter. It can be implemented by changing compilers to emit instructions that inline the algorithm with the programs instructions. It can also be implemented by using processor emulators [Boc06; QEM06] to analyze each instruction as it is emulated.

While all the above are viable implementation strategies, we have chosen to implement the algorithm with a dynamic binary re-writing tool, because this allows us to run the algorithm on unmodified binaries with reasonable performance. Specifically, we have used the Nirvana runtime instrumentation engine [BCdJ+06] to intercept, at runtime, each instruction executed by the program under analysis. Our implementation runs on Windows operating systems and Intel IA-32 processors. Nirvana performs dynamic binary translation of processor instructions, by breaking the instructions into sequences of simpler operations and optionally inserting *call* instructions to client supplied callback functions. Figure 3.4 illustrates how a simple *mov* instruction is translated. Nirvana keeps the processor state in an area called the CPU (central processing unit) context. Initially, the translated code loads into the *ecx* register a pointer to the CPU context, in order to pass it to the client callback function. After returning from the callback, the code executes the original instruction, using the state kept in the CPU context. Finally, the context is updated to reflect the CPU side-effects of the instruction. Nirvana keeps the re-written code in a code cache, thus avoiding re-translation.

```
Original instruction:
mov eax,[esp + 8]

Translated instruction:
mov ecx,cpuContext        ;pointer to CPU context maintained by nirvana
call InstructionCallback  ;calling convention assumes ecx is the first argument
mov edx, cpuContext._esp
add edx, 8
mov eax,[edx]
mov cpuContext._eax,eax   ;update CPU context
```

Figure 3.4: Example of Nirvana's translation of an IA-32 *mov* instruction.

The dynamic data-flow analysis implementation instruments every data movement instruction for Intel IA-32 [Int99] CPUs, by inserting callbacks on each of these instructions. The instrumented instructions include all variants of *mov*, *movs*, *push* and *pop* instructions. To keep track of which memory locations and CPU registers are dirty with data received from input operations, we keep a bitmap with one bit per 4K memory page, which is set if any location in the page is dirty. For every dirty page we keep an additional bitmap with one bit per memory location. We also keep an additional bitmap with a bit per CPU register to keep track of which registers are dirty. Upon receiving the callback from Nirvana, our implementation reads the current *eip* (i.e. the program counter for Intel CPUs) from the CPU state passed as the argument to the callback. Then, the implementation decodes the current instruction and updates its data structures accordingly: if the source is dirty the destination becomes dirty, otherwise it becomes clean. To bootstrap this process, whenever data is received from

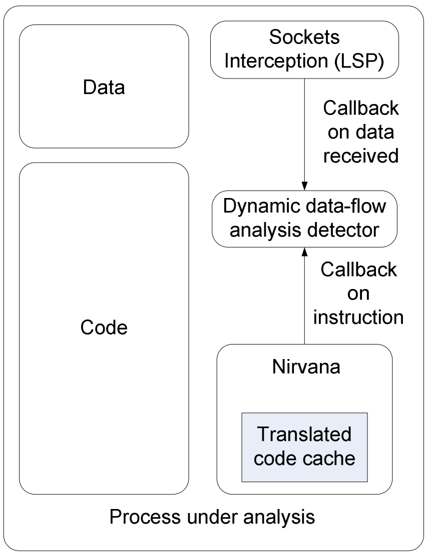

Data
Sockets Interception (LSP)
Callback on data received
Dynamic data-flow analysis detector
Callback on instruction
Code
Nirvana
Translated code cache
Process under analysis

Figure 3.5: Components inside a process running under the control of the dynamic data-flow analysis detector.

the network the memory locations where the data is written are marked dirty. To intercept network I/O, we implemented a WinSock Layered Service Provider (LSP) [HOB99]. LSPs are a simple extension mechanism for the Windows implementation of the socket interface for network programming. Finally, we also insert callbacks for every control-flow transfer instruction on IA-32 CPUs: *ret*, *call*, *jmp*, *jz*, etc, and we generate a security trap when dirty data is about to be executed or loaded into the program counter.

Figure 3.5 illustrates the components inside a process running under the control of the dynamic data-flow analysis detector. The code and data layout inside the process remain unchanged (the process may even be running when we attach the detector to it). Nirvana dynamically populates its code cache with translated instruction sequences, including callbacks to the detector code. The detector updates its data structures upon receiving callbacks and generates a security trap upon detecting an attack. Finally, it is worth pointing out that the detector code can be activated/deactivated dynamically, i.e. the process can easily switch between instrumented and non-instrumented execution.

3.2 Diversity of detection mechanisms

Vigilante can use other host-based detectors, besides dynamic data-flow analysis. We believe it is important, for several reasons, to use not only a diverse set of detection algorithms, but also different implementations of the same algorithm. Different algorithms provide different coverage and different runtime characteristics. For instance, some algorithms will be appropriate to run on production systems, while others will only be appropriate for honeypots, due to their runtime overhead. Using a diverse set of detection mechanisms makes the system more resilient to attack, because the attacker needs to successfully avoid all the detectors. Using different implementations of the same detector, makes the system more resilient to defects in the detector itself, and it also makes it more difficult for the attacker to fingerprint the detectors using the techniques described in Section 2.3.3.

Finally, it is important to point out that the self-certifying alert mechanism described in the next chapter allows detection in Vigilante to be very dynamic, for two reasons. First, it allows any host to independently decide to become a detector at any time, because detectors are not trusted. This makes it harder for an attacker to know exactly where detectors are deployed, thus making evasion more difficult. Second, it allows rapid deployment of new detection algorithms, because they don't need to be deployed at every machine in the network.

Chapter 4

Self-Certifying Alerts

Detecting a worm outbreak is not sufficient to contain it: vulnerable computers that have not yet been infected need to be protected. Vigilante enables computers to protect themselves, but first they need to be informed about the outbreak. To do this, detectors in Vigilante generate Self-Certifying Alerts (SCAs): security alerts that can be verified by the computers that receive them. Using SCAs, machines cooperate to contain an outbreak, without having to trust each other. This chapter describes the format of SCAs, as well as the mechanisms to verify, generate, and distribute alerts.

4.1 Alert types

An SCA proves that a service is vulnerable by describing how to exploit the service and how to generate an output that signals the success of the exploit unequivocally. SCAs are not a piece of code. An SCA contains a sequence of messages that, when received by the vulnerable service, cause it to reach a disallowed state. SCAs are verified by sending the messages to the service and checking whether it reaches the disallowed state. We use detection engines combined with message logging to generate SCAs at detectors.

We have developed three self-certifying alert types for Vigilante that cover the most common vulnerabilities that worms exploit:

Arbitrary Execution Control alerts identify vulnerabilities that allow worms to redirect execution to arbitrary pieces of code in a service's address space. They describe how to invoke a piece of code whose address is supplied in a message sent to the vulnerable service.

Arbitrary Code Execution alerts describe code-injection vulnerabilities. They describe how to execute an arbitrary piece of code that is supplied in a message sent to the vulnerable service.

Arbitrary Function Argument alerts identify data-injection vulnerabilities that allow worms to change the value of arguments to critical functions, for example, to change the name of the executable to run in an invocation of the `exec` system call. They describe how to invoke a specified critical function with an argument value that is supplied in a message sent to the vulnerable service.

These alert types are general. They demonstrate how the worm can gain control by using the external messaging interface to a service without specifying the low-level coding defect used to gain control. This allows the same alert types and verification procedures to be used with many different types of detection engines. Detection engine diversity reduces the false negative rate.

The three types of SCAs have a common format: an identification of the vulnerable service, an identification of the alert type, *verification information* to aid alert verification, and a sequence of messages with the network endpoints that they must be sent to during verification.

The verification information allows the verifier to craft an exploit whose success it can verify unequivocally. It is different for the different types of alerts. The verification information for an arbitrary execution control SCA specifies where to put the address of the code to execute in the sequence of messages (e.g., in which message and at which offset). Similarly, the information for arbitrary code execution SCAs specifies where to place the code to execute in the sequence of messages. Arbitrary function argument alerts have information to specify a critical function, a critical formal argument to that function, and where to put the corresponding actual argument value in the sequence of messages.

Figure 4.1 shows an example arbitrary execution control SCA generated for the Slammer worm. The SCA identifies the vulnerable service as Microsoft SQL Server version 8.00.194 and the alert type as an arbitrary execution control. The

Service: Microsoft SQL Server 8.00.194

Alert type: Arbitrary Execution Control

Verification Information: Address offset 97 of message 0

Number messages: 1

Message: 0 to endpoint UDP:1434

Message data: 04,01,01,01,01,01,01,01,01,01,01,01,01,01,01,
01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,
01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,
01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,
01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,01,DC,C9,B0,42,EB,
0E,01,01,01,01,01,01,01,70,AE,42,01,70,AE,42,...

Figure 4.1: An example arbitrary execution control SCA for the Slammer vulnerability. The alert is 457-bytes long and has been reformatted to make it human readable. The enclosed message is 376-bytes long and has been truncated.

verification information specifies that the address of the code to execute should be placed at offset 97 of message 0. The SCA also contains the 376 byte message used by the Slammer worm.

4.2 Alert verification

Verifying an SCA entails reproducing the infection process by sending the sequence of messages in the alert to a vulnerable service. It is important to run the verification procedure in a sandbox because SCAs may come from untrusted sources. The current implementation runs the verification procedure in a separate virtual machine to contain any malicious side effects. Hosts must use the same configuration to run the production instance of a service and the sandboxed instance for verification, because some vulnerabilities can be exploited only in certain program configurations.

To verify SCAs, each host runs a virtual machine with a *verification manager* and instrumented versions of network-facing services. Each service is instrumented by loading a new library into its address space with a `Verified` function that signals verification success to the verification manager. In addition, critical functions (e.g., `exec` system calls) are wrapped using a binary rewriting

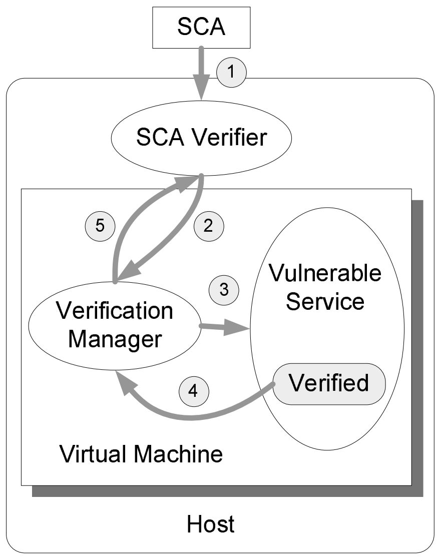


Figure 4.2: SCA verification.

tool [HB99]. The wrappers call `Verified` if the actual value of a critical argument matches a reference value specified by the verification manager. Otherwise, they call the original functions. Since we do not require access to the source code of the services, we can instrument any service. The host also runs an *SCA verifier* process outside the virtual machine that provides other processes with an interface to the verification module and acts as a reverse firewall to ensure containment.

Figure 4.2 illustrates the SCA verification procedure. When the SCA verifier receives an SCA for verification, it sends the SCA to the verification manager inside the virtual machine. The verification manager uses the data in the SCA to identify the vulnerable service. Then it modifies the sequence of messages in the SCA to trigger execution of `Verified` when the messages are sent to the vulnerable service. The modifications involve changing the byte string at the offset of the message specified in the verification information according to the alert type. This byte string is changed to:

- the address of `Verified` for arbitrary execution control alerts,
- the code for `call Verified` for arbitrary code execution alerts,

- or the reference critical argument value for arbitrary function argument alerts.

After performing these modifications, the verification manager sends the sequence of messages to the vulnerable service. If `Verified` is executed, the verification manager signals success to the SCA verifier outside the virtual machine. Otherwise, the SCA verifier declares failure after a timeout.

The state of the virtual machine is saved to disk before any verification is performed. This reference state is used to start uncompromised copies of the virtual machine for verification. After performing a verification, the virtual machine is destroyed and a new one is started from the reference state in the background to ensure that there is a virtual machine ready to verify the next SCA. The experimental results in Section 6 show that the memory and CPU overheads to keep the virtual machine running are small.

Vigilante's alert verification procedure has three important properties:

Verification is fast. The time to verify an SCA is similar to the time it takes the worm to infect the service because the overhead of the instrumentation and the virtual machine are small.

Verification is simple and generic. The verification procedure is simple and independent of the detection engine used to generate the alert. This is important for keeping the trusted computing base small, especially with many distinct detectors running in the system.

Verification has no false positives. If the verification procedure signals success, the service is vulnerable to the exploit described in the SCA. A successful verification shows that attackers can control a vulnerable service through its external messaging interface.

The current implementation has some limitations that may lead to false negatives (but not false positives). First, it assumes that the target address, code, and argument values in SCAs can be supplied verbatim in the messages that are sent during verification. This is the case in many vulnerabilities, but in others these values are transformed by the vulnerable service before being used, for example, integer values could be decoded from ASCII characters. This can potentially be addressed by specifying a conversion function for these values in SCAs.

Second, the current implementation assumes that sending the sequence of messages in an SCA to the vulnerable service is sufficient to replay the exploit during verification. This is true for all previous worms that we are aware of, but it may be insufficient for some worms. For example, the success of some exploits may depend on a particular choice of scheduling order for the threads in a service. We could address this limitation by including other events in SCAs (e.g., scheduling events and other I/O events) and by replaying them during verification. There is a large body of work in this area [EAWJ02; DKC+02] that we could leverage.

4.3 Alert generation

Hosts generate SCAs when they detect an infection attempt by a worm. Vigilante enables hosts to use any detection engine provided it generates an SCA of a supported type. SCA generation follows the same general pattern for all detection engines and services, but some details are necessarily detection engine specific.

To generate SCAs, hosts log messages and the networking endpoints where they are received during service execution. We garbage collect the log by removing messages that are included in generated SCAs or that are blocked by our filters. We also remove messages that have been in the log more than some threshold time (e.g., one hour).

When the engine detects an infection attempt, it searches the log to generate candidate SCAs and runs the verification procedure for each candidate. The strategy to generate candidate SCAs is specific to each detection engine, but verification ensures that an SCA includes enough of the log to be verifiable by others and it filters out any false positives that detectors may generate. SCA generation returns a candidate SCA when that SCA passes verification.

We implemented SCA generation for two detection engines: the non-executable (*NX*) pages [PAX01] algorithm, which we describe next, and the dynamic data-flow analysis detection algorithm described in Chapter 3. We chose these engines because they represent extreme points in the trade-off between coverage and overhead: the first detector has low overhead but low coverage whereas the second has high overhead and high coverage. Furthermore, they are both widely applicable, since neither of them requires access to source code.

4.3.1 Using Non-executable pages

The first detection engine uses non-execute protection on stack and heap pages to detect and prevent code injection attacks. It has negligible runtime overhead with emerging hardware support and has relatively low overhead even when emulated in software [PAX01]. This detector can be used to generate arbitrary execution control or arbitrary code execution SCAs as follows.

When the worm attempts to execute code in a protected page, an exception is thrown. The detector catches the exception and then tries to generate a candidate SCA. First, the detector traverses the message log from the most recently received message searching for the code that was about to be executed or for the address of the faulting instruction. If the detector finds the code, it generates a candidate arbitrary code execution SCA, and if it finds the address of the faulting instruction, it generates a candidate arbitrary execution control SCA. In both cases, the message and the offset within the message are recorded in the verification information, and the single message is inserted in the candidate SCA.

The detector then verifies the candidate SCA. Since most worms exploit vulnerabilities using only one message to maximize their propagation rate, this candidate SCA is likely to verify. However, it will fail verification for multi-message exploits. In this case, the detector includes additional messages by taking longer suffixes of the message log and including them in the candidate SCA. The detector keeps increasing the number of messages in the candidate SCA until the SCA verifies or the message log is empty.

The search through the log is efficient when detectors are run in honeypots because the detection engine will receive only anomalous traffic and the message log will be small. We optimize for this case by including all the logged messages in the first candidate SCA when the log size is smaller than a threshold (e.g., 5).

4.3.2 Using dynamic data-flow analysis

Dynamic data-flow analysis can be used to generate the three types of alerts discussed in Section 4.1. By tracking the flow of data received from the network, dynamic data-flow analysis can generate efficiently the verification information needed for SCAs. To do this, the instrumented data movement instructions are used to maintain data structures that indicate not only which CPU registers and memory locations are dirty but also where the dirty data came from. Each dirty register and memory location has an associated integer that identifies the input message and offset where the dirty data came from. These identifiers are simply a sequence number for every byte received in input messages. There is a bitmap with one bit per 4K memory page; the bit is set if any location in the page is dirty. For each page with the bit set, an additional table is maintained with one identifier per memory location. We also keep a table with one identifier per CPU register. Finally, we keep a list with the starting sequence number for every input message to map identifiers to messages.

The modified dynamic data-flow algorithm proceeds in a manner similar to the one described in Section 3.1.1: whenever an instruction that moves data from a source to a destination is executed, the destination becomes dirty if the source is dirty and becomes clean otherwise. When a destination becomes dirty, it is tagged with the identifier associated with the source. Whenever data is received from a network connection, the memory locations where the data is written are marked dirty and tagged with sequence numbers corresponding to each received byte. The instrumented control flow instructions signal an infection attempt when dirty data is about to be executed or loaded into the program counter, while the instrumented critical functions signal an infection attempt when all the bytes in a critical argument are dirty. The algorithm generates a candidate SCA of the appropriate type when it detects an infection attempt:

- If dirty data is about to be loaded into the program counter, it signals an attempt to exploit an arbitrary execution control vulnerability.
- If dirty data is about to be executed, it signals an attempt to exploit an arbitrary code execution vulnerability.

- If a critical argument to a critical function is dirty, it signals an attempt to exploit an arbitrary function argument vulnerability.

The additional information maintained by this engine eliminates the need for searching through the log to compute the verification information: this information is simply read from the data structures maintained by the engine. The identifier for the dirty data is read from the table of dirty memory locations or the table of dirty registers. The identifier is mapped to a message by consulting the list of starting sequence numbers for input messages and the offset in the message is computed by subtracting the starting sequence number from the identifier. Then, the detector adds the single identified message to the candidate SCA and attempts to verify it. This verification will succeed for most worms and it completes the generation procedure. For multi-message exploits, the detector follows the same search strategy to compute candidate SCAs as the detector based on non-executable pages.

We will use the vulnerable code in Figure 4.3 to illustrate SCA generation using dynamic data-flow analysis (the source code for the program is shown in Figure 2.1), during an edge injection attack. When the code starts to execute, the *ebx* register holds the *message* parameter. The parameter points to a message just received from the network. In this example, the bytes in the incoming attack message were mapped to identifiers from 100 to 127. Before the code is executed, the memory region where the message was received is marked dirty with identifiers from 100 to 127. The code starts by doing a range check on the first byte of the message, by subtracting `0x10` and `0x31` from the first byte in the message and then comparing the result with a constant (`0x0E`). If the check succeeds, the next bytes in `message` are copied to a stack-based buffer until a newline character is found. This results in a buffer overflow that overwrites the return address on the stack. After running the range check on the first byte of the message, on line 8, the program loads the second byte of the message into the *dl* register, on line 10. At this point *dl* is marked dirty and tagged with identifier 101. The function then enters a loop, on lines 21 to 26, that copies the first field of the message into the *request* buffer. When instruction 21 executes, the memory location pointed to by *eax* is marked dirty and tagged with identifier 101, since *dl* is also tagged with 101.

```
 1: push     ebp                     ;on entry, ebx points to the message parameter
                                     ;esp points to eip saved on the stack
                                     ;the memory containing the message is tagged with
                                     ;identifiers 100 to 127
 2: mov      ebp,esp
 3: mov      al,byte ptr [ebx]       ;move first byte of message into al
                                     ;tag al with 100
 4: mov      ecx,dword ptr [ebp+8]
 5: sub      al,10h
 6: sub      al,31h
 7: sub      esp,8                   ;allocate stack space for request buffer
 8: cmp      al,0Eh                  ;perform range check on first byte
 9: ja       45
10: mov      dl,byte ptr [ebx+1]     ;move second byte of message into dl
                                     ;tag dl with 101
11: push     esi
12: push     edi
13: lea      edi,[ebx+1]             ;move address of second byte into edi
14: xor      esi,esi
15: cmp      dl,0Ah
16: lea      eax,[ebp-8]             ;move address of request buffer into eax
17: je       28
18: mov      ecx,eax
19: sub      edi,ecx
20: lea      esp,[esp+0h]

21: mov      byte ptr [eax],dl       ;copy next byte into request buffer
                                     ;tag address pointed to by eax with 100+i
22: mov      dl,byte ptr [edi+eax+1] ;move next byte of message into dl
                                     ;tag dl with 100+i
23: add      eax,1
24: add      esi,1
25: cmp      dl,0Ah                  ;if not found 0A, continue to next byte.
26: jne      21

...                                  ;irrelevant instructions omitted

48: mov      esp,ebp
49: pop      ebp
50: ret                              ;load value pointed to by esp into eip
                                     ;generate an SCA, because
                                     ;esp points to dirty memory
```

Figure 4.3: Example of SCA generation with vulnerable program in IA-32 assembly language (compiled from the source code in Figure 2.1).

Instruction 22, loads the next byte of the message into *dl*, which becomes tagged with 102. The byte is then compared with the newline character (*0x0A*), and the loop continues if the newline was not reached. The loop eventually overwrites the stored return address.

Figure 4.4 shows the state of memory before and after the vulnerable code is executed. When the `ret` instruction is about to execute, at the end of the

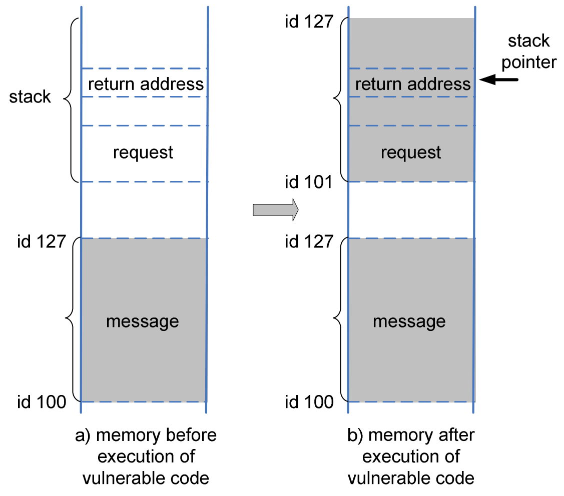


Figure 4.4: Example of SCA generation with dynamic data-flow analysis. The figure shows the memory when (a) a message is received and the vulnerable code is about to execute, and (b) after the vulnerable code executes and overwrites the return address in the stack. Greyed areas indicate dirty memory regions and the identifiers of dirty data are shown on the left.

function, a portion of the stack has been marked dirty with identifiers from 101 to 127 because the instrumented data movement instructions propagated the tags from the message into the stack buffer, while copying the message data. Since the copy overwrote the return address in the stack, the `ret` instruction attempts to load dirty data into the program counter. Therefore, the detector generates an arbitrary execution control alert: it computes the verification information from the identifier of the dirty data pointed to by the stack pointer and adds the identified message to the SCA. This message is the attack message because the identifier of the dirty data falls in the range 100 to 127, and the offset is computed by subtracting 100 from the identifier. The detector verifies this SCA and sends it to the distribution and protection modules.

As explained in Chapter 3, dynamic data-flow analysis suffers from a small but non-negligible false positive rate. It also has a substantial runtime overhead, when implemented with dynamic binary re-writing. SCAs address both of these issues: verification eliminates false positives and the cooperative detection architecture spreads the detection load.

4.4 Alert distribution

After generating an SCA, a detector broadcasts it to other hosts. This allows other hosts to protect themselves if they run a program with the vulnerability in the SCA.

The mechanism to broadcast SCAs must be fast, scalable, reliable and secure. It must be fast because there is a race between SCA distribution and worm propagation. Scalability is a requirement because the number of vulnerable hosts can be extremely large. Additionally, SCA distribution must be reliable and secure because the growing number of hosts compromised by the worm can launch attacks to hinder distribution and the number of detectors sending an SCA for a particular vulnerability can be small. The SCA must be delivered to vulnerable hosts with high probability even under these extreme conditions. To meet these requirements, Vigilante uses a secure Pastry overlay [CDG+02] to broadcast SCAs.

Vigilante uses flooding to broadcast SCAs to all the hosts in the overlay: each host sends the SCA to all its overlay neighbours. Since the overlay is scalable, we can distribute an SCA to a large number of hosts with low delay in the absence of attacks. Each host maintains approximately $15 \times log_{16}N$ neighbours and the expected path length between two hosts is approximately $log_{16}N$. Since each host has a significant number of neighbours, flooding provides reliability and resilience to passive attacks where compromised hosts simply refuse to forward an SCA. Hosts that join the overlay can obtain missing SCAs from their neighbours.

The secure overlay also includes defences against active attacks. It prevents sybil attacks [Dou02] by requiring each host to have a certificate signed by a trusted offline certification authority to participate in the overlay [CDG+02]. The certificate binds a random *hostId* assigned by the certification authority with a public key whose corresponding private key should be known only to the host. This prevents attackers from choosing their identifiers or obtaining many identifiers because these keys are used to challenge hosts that want to participate in the overlay.

Additionally, the secure overlay prevents attackers from manipulating the overlay topology by enforcing strong constraints on the hostIds of hosts that can be overlay neighbours [CDG+02]. These constraints completely specify the set of neighbours of any host for a given overlay membership. Each host establishes authenticated and encrypted connections with its neighbours using the certified public keys. Since compromised hosts cannot choose their hostIds, they are not free to choose their neighbours and they are not able to increase the number of overlay paths through compromised hosts.

Compromised hosts in the overlay may also attempt to disrupt SCA distribution with denial of service attacks. Vigilante uses three techniques to mitigate these attacks: hosts do not forward SCAs that are blocked by their filters or are identical to SCAs received recently, they only forward SCAs that they can verify, and they impose a rate limit on the number of SCAs that they are willing to verify from each neighbour. The first technique prevents attacks that flood variants of old SCAs and the second prevents attacks that flood bogus SCAs to all the hosts in the overlay. Since hosts only accept SCAs received over the authenticated connections to their neighbours, the third technique bounds the computational overhead that compromised hosts can impose on their neighbours. It is effective because the constraints on neighbour identifiers make it hard to change neighbours.

Requiring hosts to verify SCAs before forwarding raises some issues. Some hosts may be unable to verify valid SCAs because they do not have the vulnerable software or they run a configuration that is not vulnerable. We made overlay links symmetric to reduce the variance in the number of neighbours per host and to ensure that there is a large number of disjoint overlay paths between each pair of nodes. Since flooding explores all paths in the overlay, the probability that SCAs are delivered to vulnerable nodes is very high even when the fraction of nodes that can verify the SCA is small.

Additionally, verifying SCAs introduces delay. Our verification procedures are fast but the attacker can increase delay with denial of service attacks. In addition to the techniques above, we verify SCAs from different neighbours concurrently to defend against attacks that craft SCAs that take a long time to verify. Therefore, the attacker can increase the verification delay at a host by a factor proportional to the number of compromised neighbours of the host.

Most worms have propagated by randomly probing the IP address space but they could propagate much faster by using knowledge of the overlay topology. Therefore, it is important to hide information about the overlay topology from the worm. One technique to achieve this is to run the overlay code in a separate virtual machine and to enforce a narrow interface that does not leak information about the addresses of overlay neighbours.

Our preferred technique to hide information about the overlay topology from the worm is to run an overlay with super-peers. The super-peers are not vulnerable to most worm attacks because they run only the overlay code and a set of virtual machines with sandboxed versions of vulnerable services to verify SCAs efficiently. The super-peers form a secure Pastry overlay as we described. Each ordinary host connects to a small number q of super-peers (e.g., $q = 2$) that are completely specified by the host's identifier. This prevents leaking information about vulnerable hosts because all neighbours of compromised hosts are super-peers that do not run vulnerable software.

An overlay with super-peers is also more resilient to denial of service attacks. First, we can give priority to verification of SCAs sent by super-peers. Since super-peers are less likely to be compromised than ordinary hosts, this is a very effective defence against denial of service attacks that bombard hosts with SCAs. Additionally, super-peers may be well connected nodes with very large link capacities to make it hard for attackers to launch denial of service attacks by simply flooding physical links.

Currently, a secure overlay with super-peers is the best option for deployment of SCA distribution. It could be supported easily by an infrastructure similar to Akamai's, which is already used by anti-virus companies to distribute signatures [Aka00]. However, it should be noted that alerts could be distributed over other broadcast/multicast channels (e.g., channels used to broadcast video).

4.5 Implementation

The implementation of SCA generation uses techniques similar to the ones described in Chapter 3, for the implementation of the dynamic data-flow analysis detector. Vigilante intercepts socket operations, using a Layered Service Provider [HOB99], to log received messages and to mark the socket buffers dirty. Each new byte received is tagged with a unique 32 bit identifier. Tags are propagated when dirty data moves across memory and registers, by using Nirvana [BCdJ$^+$06] to translate code sequences dynamically into instrumented versions. This instrumentation ensures that the detection engine is invoked before every instruction to disassemble the instruction, examine its operands, and update the data structures that keep track of dirty data. These data structures are similar to the ones described in Chapter 3, for the implementation of the dynamic data-flow analysis detector, except that they store 32 bit identifiers for dirty data, instead of single bits. When a control transfer instruction is about to give control to the worm, the engine generates an SCA from these data structures and the message log (as described in Section 4.3.2).

SCAs are verified inside a Virtual PC 2004 virtual machine (VM) to isolate any side-effects of the verification process (see Figure 4.2). During an initial setup phase, the SCA verifier process starts a VM and establishes a virtual network connection to the verification manager inside the VM. The verification manager initiates the connection because the VM is configured to disallow any incoming connections. The SCA verifier then instructs the verification manager to load network facing services. The verification manager injects a dynamic link library (DLL) into each service by creating a new thread that loads the DLL. The DLL includes the `Verified` function and an initialization routine which reports the address of the `Verified` function back to the verification manager, through a shared memory section. At this stage the setup for verification is complete and the virtual machine state is saved.

When an SCA arrives, the SCA verifier relays the SCA to the verification manager, sets a timer, and waits for a success notification message or the timeout. The verification manager replays the messages in the SCA, using the address of the Verified function as described in Section 4.2, and waits on a synchronization object. If the SCA is valid, the `Verified` function is called and sets the synchronization object, signalling success to the verification manager, who sends a success notification message to the SCA verifier. After each verification, the VM is destroyed and a new one is created from the state on disk to be ready to verify the next SCA.

The implementation of the overlay used for distribution is described in [CCR04; CDG+02]. We used a small real network to evaluate the distribution of SCAs. To understand the behaviour of Vigilante on the Internet, we simulated the distribution system using topologies from the secure version of the overlay [CDG+02]. The simulations also used measurements from real worm outbreaks and from our implementation of Vigilante.

Chapter 5

Protection

The last crucial step to contain a worm outbreak is to protect vulnerable computers that have not been infected yet. After receiving an SCA for the outbreak, vulnerable hosts protect themselves, but first they verify the SCA, to prevent false positives. If the verification is successful, the local version of the program, with the local configuration, is vulnerable to the exploit described in the SCA. If the verification fails, the SCA is dropped and the host does not consume more resources with the protection procedure. This is important for mitigating denial-of-service attacks because verification is significantly cheaper than generating protective countermeasures.

After successful verification of the SCA, hosts could stop the vulnerable program or run it with a detection engine to prevent infection. However, stopping the program is not acceptable in most settings and running a high-coverage detection engine (e.g., dynamic data-flow analysis) results in poor performance. Additionally, detection engines typically detect the infection attempt too late for the vulnerable program to be able to recover gracefully.

Instead, hosts in Vigilante generate filters to block worm traffic before it is delivered to the vulnerable program, and they suspend the vulnerable program to prevent infection during the filter generation process. Once generated, these filters allow the program to continue running while under attack. Furthermore, they are unlikely to affect the correct behaviour of the program, since they do not change the program's code; they just discard attack messages. The main challenge in generating these filters is to make them block mutations of the worm attack. In this chapter we describe the optimal filters and present an algorithm to automatically generate filters that are effective at blocking mutations of worm traffic, have no false positives, and introduce very low overhead.

5.1 Sufficient preconditions for infection

The optimal filter for a worm blocks all mutations of attack messages and has no false positives. This filter can be expressed in terms of *weakest preconditions*, as defined by Dijkstra [Dij75]. We assume a system that processes input messages by running a vulnerable program P, instrumented to terminate when it reaches a state satisfying the condition I that defines successful infection. The optimal filter for this system, in regard to I, is the weakest precondition for infection, i.e. the weakest condition which is guaranteed to lead to an infected state:

$$wp(P, I)$$

While of theoretical interest, calculating weakest preconditions is currently not practical for most real systems [Win93]. However, it is practical to generate filters that capture *sufficient preconditions* for infection: a set of conditions on attack messages such that there are program states and scheduling decisions for which the messages satisfying these conditions are guaranteed to lead to successful infection. This means that when an entity, malicious or not, sends a message satisfying these conditions, the message may lead to successful infection; therefore, we classify it as an attack message. The filters generated automatically by Vigilante have no false positives, because they only drop attack messages.

5.2 Vulnerability condition slicing

5.2.1 Algorithm

Hosts generate the conditions for filters automatically by analyzing the execution path followed when the messages in the SCA are replayed. They use *vulnerability condition slicing*: a form of dynamic data and control flow analysis that finds the conditions on the messages in the SCA that determine the execution path that exploits the vulnerability.

The dynamic data-flow analysis during filter generation is more elaborate than the one we use to detect worms. It instruments all instructions in the program to compute data-flow graphs for dirty data, i.e., data derived from the messages in the SCA. These data-flow graphs describe how to compute the current value of the dirty data: they include the instructions used to compute the current value from the values at specified byte offsets in the messages and from constant values read from clean locations. We associate a data-flow graph with every memory position, register, and processor flag that stores dirty data.

The control-flow analysis keeps track of all conditions that determine the program counter value after executing control transfer instructions (conditional move and set instructions are handled similarly to control transfer instructions, therefore we omit them for brevity). We call the conjunction of these conditions the *filter* condition. The filter condition is initially *true* and it is updated after every instruction that uses a dirty processor flag or transfers control to an address read from a dirty location. The filter condition is updated to be the conjunction of its old value and the appropriate conditions on the expressions computed by the data-flow graphs of the dirty flag and address location.

Figure 5.1 shows the vulnerability condition slicing algorithm in pseudo-code. When the program receives a message, the algorithm tags the memory positions where each byte in the message is stored with a new data-flow graph that identifies the byte (input bytes are identified by an increasing *counter*). Whenever an instruction is executed, the algorithm checks if its arguments are tagged with data-flow graphs. If so, the address that stores the result of the instruction is tagged with a new data-flow graph reflecting the execution of the instruction; otherwise the address that stores the result is marked clean. If the instruction affects the processor's flags, they are tagged in a similar fashion.

When a conditional control-flow instruction is executed, the flag controlling the instruction is checked for dirtiness. If it is dirty, the filter condition is updated to reflect the conditions tested by the instruction and outcome of the check on the flag. This is done by creating a new data-flow graph that applies the opcode of the instruction to the data-flow graph of the flag controlling the jump; the

```
INITIALIZE
    $filter$ = GRAPH(TRUE);
    $counter = 0$

ONRECEIVEDNETWORKMESSAGE($address$, $size$)
    for $i \leftarrow 0$ to $size - 1$
          do
              SETTAG($address + i$, GRAPH(INPUT, $counter$))
              $counter = counter + 1$


GETTAGORVALUE($a$)
    if ISDIRTY($a$)
        then return GETTAG($a$)
        else  return GRAPH(CONSTANT, VALUEAT($a$))


ONINSTRUTION($address$, $opcode$, $a$, $b$)
    if ISDIRTY($a$) or ISDIRTY($b$)
        then TAG($address$, GRAPH($opcode$, GETTAGORVALUE($a$), GETTAGORVALUE($b$)))
              foreach $flag$ in CHANGEDFLAGS($opcode$)
                    do TAG($flag$, GRAPH($opcode$, GETTAGORVALUE($a$), GETTAGORVALUE($b$)))

        else  CLEAR($address$)
              foreach $flag$ in CHANGEDFLAGS($opcode$)
                    do CLEAR($flag$)


ONCONDITIONALCONTROLFLOWTRANSFER($flag$, $opcode$, $taken$)
    if ISDIRTY($flag$)
        then
              if $taken$
                 then $filter$ = GRAPH(AND, $filter$, GRAPH($opcode$, GETTAG($flag$)))
                 else  $filter$ = GRAPH(AND, $filter$, GRAPH($\neg opcode$, GETTAG($flag$)))


ONINDIRECTCONTROLFLOWTRANSFERORINDIRECTMEMORYACCESS($address$)
    if ISDIRTY($address$)
        then $filter$ = GRAPH(AND, $filter$, GRAPH(EQUAL, VALUEAT($address$), GETTAG($adress$)))
```

Figure 5.1: Vulnerability condition slicing algorithm. The algorithm generates filters that block mutations of worm attacks, by analyzing a vulnerable program and extracting the control-flow decisions that lead to successful attacks.

outcome of the test on the flag is recorded by negating the opcode if the jump is not *taken*. When an indirect control-flow transfer uses a dirty location (memory or register), the filter condition is updated to reflect that the data-flow graph for the dirty location must be *equal* to the current value stored there. The filter is updated similarly on indirect memory accesses which use a dirty address operand.

For example, when the instruction `jz address` (jump if zero to address) is executed, the filter condition is left unchanged if the zero flag is clean. If the zero flag is dirty and the jump is taken, we add the condition that the expression computed by the data-flow graph for the zero flag be true. If the zero flag is dirty and the jump is not taken we add the condition that the expression computed by the data-flow graph for the zero flag be false. As another example, when `jmp eax` (jump to the memory position identified by the *eax* register) is executed, the filter condition is left unchanged if the `eax` register is clean. If `eax` is dirty, we add the condition that the expression computed by `eax`'s data-flow graph be equal to the value currently stored by `eax`.

We will use the vulnerable code in Figure 5.2, and the corresponding arbitrary execution control SCA from Section 4.3, to illustrate the filter generation procedure (the mechanics of attacks on this code were described in Chapter 2). When the code starts to execute, the *ebx* register holds the `message` parameter. The parameter points to a message just received from the network. Before the code is executed, the memory region where the message was received is tagged with data-flow graphs with symbols `input[0]` to `input[27]`, corresponding to the bytes just received in the message. The code starts by loading the first byte of the message into *al*; at this point *al* is tagged with `input[0]`. Next, the code does a range check on the first byte of the message by subtracting `0x10` and `0x31` from it, and comparing the result with `0x0E`. Thus, at instruction 6, *al* becomes tagged with `input[0] - 0x10 - 0x31`. The zero, sign and overflow flags become dirty after the comparison at instruction 8, and their data-flow graphs become `input[0] - 0x10 - 0x31`=`0x0E`; Figure 5.3 a) shows the data-flow graph associated with the flags at this point. The filter condition is updated to `input[0] - 0x10 - 0x31`<=`0x0E` after instruction 9, because the conditional jump *ja* is not taken.

```
 1: push    ebp                   ;on entry, ebx points to the message parameter
                                  ;esp points to eip saved on the stack
                                  ;the memory containing the message is tagged with
                                  ;data-flow graphs for symbols input[0] to input[27]
 2: mov     ebp,esp
 3: mov     al,byte ptr [ebx]     ;move first byte of message into al
                                  ;tag al with input[0]
 4: mov     ecx,dword ptr [ebp+8]
 5: sub     al,10h                ;tag al with input[0] - 0x10
 6: sub     al,31h                ;tag al with input[0] - 0x10 - 0x31
 7: sub     esp,8                 ;allocate stack space for request buffer
 8: cmp     al,0Eh                ;perform range check on first byte
                                  ;tag flags with input[0] - 0x10 - 0x31 = 0x0E
 9: ja      45                    ;add filter condition input[0] - 0x10 - 0x31 <= 0x0E
                                  ;because the jump is not taken
10: mov     dl,byte ptr [ebx+1]   ;move second byte of message into dl
                                  ;tag dl with input[1]
11: push    esi
12: push    edi
13: lea     edi,[ebx+1]           ;move address of second byte into edi
14: xor     esi,esi
15: cmp     dl,0Ah                ;tag flags with input[1] = 0x0A
16: lea     eax,[ebp-8]           ;move address of request buffer into eax
17: je      28                    ;add filter condition input[1]!= 0x0A
                                  ;because the jump is not taken
18: mov     ecx,eax
19: sub     edi,ecx
20: lea     esp,[esp+0h]

21: mov     byte ptr [eax],dl     ;copy next byte into request buffer
                                  ;tag address pointed to by eax with input[i]
22: mov     dl,byte ptr [edi+eax+1] ;move next byte of message into dl
                                  ;tag dl with input[i]
23: add     eax,1
24: add     esi,1
25: cmp     dl,0Ah                ;tag flags with input[i] = 0x0A
26: jne     21                    ;add filter condition input[i] != 0x0A
                                  ;because the jump is taken
```

Figure 5.2: Example of filter generation with vulnerable program in IA-32 assembly language (compiled from the source code in Figure 2.1).

The function then copies bytes from the message into the *request* buffer, until it finds the terminator character 0x0A. The check for termination on the second byte is implemented by instructions 15 and 17, and the remaining bytes are checked by instructions 25 and 26. For each iteration of the copy loop, the *dl* register holds the next byte in the message, and is therefore tagged with `input[i]` (for `i` >= `1`). Each iteration adds a filter condition of the form `input[i]`≠`0x0A` for `i` >= `1`, because *dl* is compared with `0x0A` and a conditional jump continues the loop if they are not equal; Figure 5.3 b) shows the data-flow graph associated with the flags when these control-flow decisions are taken.

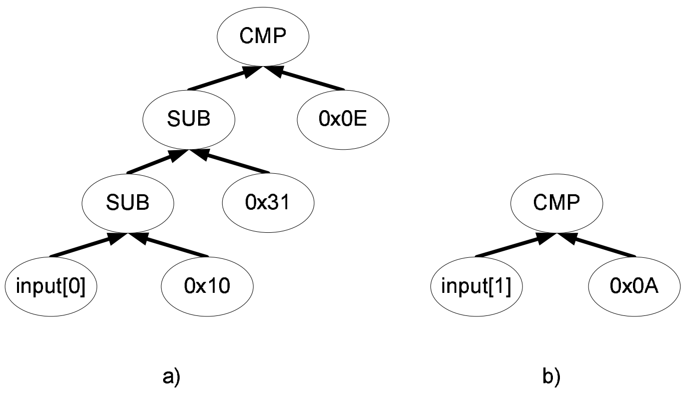


Figure 5.3: Data-flow graphs for flags controlling conditional jumps: a) when the instruction `ja 45` is executed, and b) when the instruction `jne 21` is executed. Both instructions are executed by the vulnerable program in Figure 5.2.

Figure 5.4 shows the filter condition generated by the algorithm for this example. It shows that the algorithm generalizes the attack by noting that messages will lead to successful attacks if they have a first byte within the appropriate range and a sufficient number of subsequent bytes different from the newline character.

The termination condition for the filter generation procedure depends on the type of SCA. The filter generation procedure replays the execution triggered by receiving the message in the SCA after updating the location specified by the verification information to contain a *verification nonce*. The idea is to use the dynamic data-flow analysis to stop execution in the same conditions that we described for detection while using the verification nonce to prevent false positives. For example, the filter generation procedure for arbitrary code execution alerts stops when the program is about to jump to the nonce value. To remove unnecessary conditions from the filter, the generation procedure returns the value of the filter condition after the instruction that overwrites the critical argument or jump target that causes the worm to gain control. To obtain the value of the filter condition at this point, we tag write operations with the current value of the filter condition.

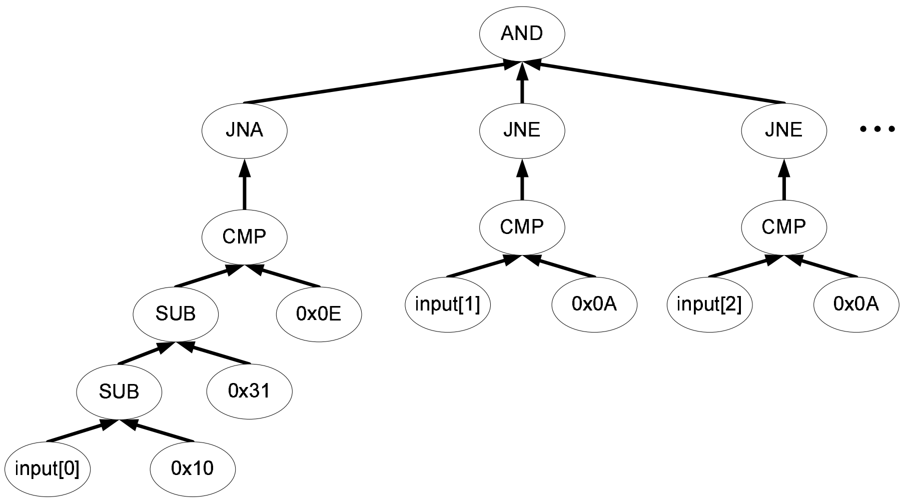


Figure 5.4: Filter condition for an edge injection attack on the program in Figure 5.2. The filter blocks mutations of the attack; it matches any attack message with the first byte in the allowed range and the subsequent bytes different from 0x0A up until the bytes that overwrite the return address on the stack.

The filters generated by this algorithm are safe. The conditions generated by the algorithm can be computed without propagating side-effects to memory or the processor, because they are pure functional expressions. In addition, the filter conditions do not include loops or recursion. Therefore, they can always be computed in linear time or less, on the size of the corresponding data-flow graphs. Figure 5.5 shows the translation of the filter condition in Figure 5.4 into a filtering program. The translation is carried out by doing a depth-first traversal of the graph to generate a stack-based evaluation of the data-flow expression. We ensure that the code generated has no side effects, by saving/restoring the CPU state when entering/leaving the filter code and by using a separate stack that we ensure is large enough to evaluate the data-flow expressions. Filters also check that a message is at least as long as the largest offset used by the filter code.

Filters generated using this procedure have no false positives: any message that matches the filter condition would be able to exploit the vulnerability if received in the state in which the filter was generated, and if scheduling decisions were identical. Additionally, they can filter many worm variants that exploit the same vulnerability because the filter captures the exact conditions that determine the path to exploit the vulnerability. These filters are very different from filters that block messages that contain a particular string [KK04; SEVS04] or sequence of strings [NKS05]. They can capture arbitrary computations on the values of the input messages.

This algorithm can be seen a form of program slicing [Wei84]. It identifies a subset of instructions in the program that compute the control-flow decisions that lead to successful attacks. The instructions captured in the data-flow graphs in Figure 5.3 are a subset of the instructions of the vulnerable program shown in Figure 2.2. Filters block messages that satisfy these conditions, by computing the conditions immediately after messages are received.

The algorithm can also be seen a form of symbolic execution [Kin76]: simultaneously with the concrete execution of the vulnerable program, the algorithm executes symbolically the instructions that process dirty data.

The current implementation only supports filters with conditions on a single message. To deal with SCAs with multiple messages in their event list, we produce a filter that blocks a critical message in the list to prevent the attack. The filter is obtained using the generation procedure that we described above and removing all conditions except those related to the critical message. We pick this critical message to be the one named in the SCA's verification information because this is the message that carries the worm code or the value used to overwrite a control structure or a critical argument. To prevent false positives, we only install the filter if this is also the message that gives the worm control when it is processed.

The filters that we described so far have no false positives but they may be too specific. They may include conditions that are not necessary to exploit the vulnerability. For example, the filter generated for the Slammer worm would

```
mov eax, message_len           ;move the message length into eax
cmp eax, 14                    ;check maximum index used in filter conditions
jb do_not_drop                 ;if message is shorter, do not drop it

mov esi, message               ;move address of message into esi
xor eax,eax                    ;clear eax register
mov al,byte ptr [esi + 0x00]   ;move first byte into al
push eax
push 0x10
pop ebx
pop eax
sub al,bl                      ;subtract 0x10 from al
push eax
push 0x31
pop ebx
pop eax
sub al,bl                      ;subtract 0x31 from al
push eax
push 0x0E
pop ebx
pop eax
cmp al, bl                     ;compare al with 0x0E
ja do_not_drop                 ;if above, do not drop the message

xor eax,eax                    ;clear eax register
mov al,byte ptr [esi + 0x01]   ;move second byte into al
push eax
push 0x0A
pop ebx
pop eax
cmp al,bl                      ;compare with 0x0A
je do_not_drop                 ;if second byte is 0x0A, do not drop the message

...                            ;the remaining bytes, until the ones
                               ;that overwrite the return address on the stack,
                               ;are also checked to be different from 0x0A
```

Figure 5.5: Filter code generated automatically for the filter condition in Figure 5.4. The filter blocks mutations of an edge injection attack on the vulnerable program shown in Figure 5.2. The code to save registers and to setup a separate stack is omitted for brevity.

require a longer than necessary sequence of non-zero bytes. This filter would not block variants of the worm that used smaller messages.

We use two filters to reduce false negatives while ensuring that we have no false positives: a *specific filter* without false positives, and a *general filter* that may have false positives but matches more messages than the specific filter to block more worm variants.

Messages are first matched against the general filter. If a message does not match, it is sent to the program for immediate processing. Otherwise, it is matched against the specific filter. A message that matches is dropped and one that does not is sent to a dynamic data-flow analysis detection engine. If the engine determines that the message is innocuous, it is sent to the program for processing. But if the engine detects an attempt to exploit a vulnerability, the message is dropped after being used to generate an SCA. This SCA can be used to make the specific filter more general: the specific filter's condition can be updated to be the disjunction of its old value and the filter condition generated from the SCA using the procedure from the previous section.

Since detection with dynamic data-flow analysis is expensive, the general filter must have a low false positive rate for the protected program to achieve good performance. We create the general filter by removing some conditions from the specific filter using heuristics guided by information about the structure of the path that exploits the vulnerability.

The first heuristic removes conditions on message bytes that appear after the offset identified by the verification information in the SCA. Since the bytes in the message are usually processed in order, this heuristic is unlikely to introduce false positives. The second heuristic removes conditions added by the execution of a function when that function returns. The rationale is that these conditions are usually not important after the function returns and that the important effects of the function are captured in the data-flow graphs of dirty data. The third heuristic removes conditions added by indirect memory accesses, since they may unnecessarily constrain inputs due to coding idioms used in common implementations of runtime libraries. We compute the general filter at the same time as the specific filter by maintaining a separate *general filter* condition to which we apply these heuristics. Our experimental results suggest that these heuristics work well in practice: they generalize the filter to capture most or even all worm variants and they appear to have zero false positives.

5.2.2 Implementation

The implementation of filter generation uses techniques similar to the ones described in Chapter 3, for the implementation of the dynamic data-flow analysis detector. We associate a data-flow graph with every memory position, register, and processor flag that stores dirty data. We maintain a page table with one entry per 4K memory page; if any byte in the page is dirty, the entry points to a table with one pointer per memory location. If a location is dirty, the corresponding entry in this table points to a data-flow graph. A separate data structure stores data-flow graphs for registers and flags.

The implementation intercepts socket operations, and tags each received byte with a unique data-flow graph that identifies the byte. We use Nirvana [BCdJ+06] to instrument all IA-32 instructions to maintain the data-flow graphs up to date. These data-flow graphs describe how to compute the current value of the dirty data: they include the instructions used to compute the current value from the values at specified byte offsets in input messages and from constant values read from clean locations. In the current implementation, each data-flow graph has constants, byte offsets in messages, and Intel IA-32 opcodes as vertices and the edges connect the operands of an instruction with its opcode. The filter condition is represented as a list of graphs with the same format. Therefore, the filter condition can be translated into efficient executable IA-32 code for filtering incoming messages, as shown in Figure 5.5. Furthermore, we ensure that the filter code has no side effects and that it always terminates, since it includes only forward jumps.

After filters are generated, we deploy them with the Detours [HB99] runtime instrumentation package. The interception mechanism used by Detours has very low overhead, therefore it is appropriate for use in production systems, where the filters will be deployed. Deploying the filters on a vulnerable host does not require re-starting the vulnerable service. To achieve hot installation of the filters, the functions that intercept the socket interface check for availability of filters on a shared memory section. After filter generation, the filter code is copied to the vulnerable process through the shared memory section. Figure 5.6 shows the components inside a vulnerable process, after a filter is deployed.

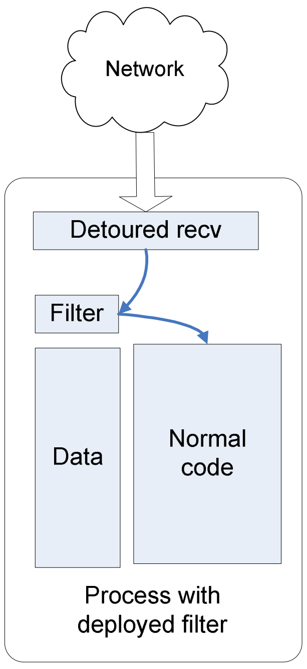


Figure 5.6: Components inside a process with a filter deployed by Vigilante. Vigilante intercepts socket functions to process network messages with the filter code. When the filter matches a message, it is dropped, otherwise it is handed over to the normal code.

Chapter 6

Evaluation

We implemented a prototype of Vigilante for Intel IA-32 machines running the Windows operating system. This section evaluates our implementation of the Vigilante algorithms and architecture.

6.1 Experimental setup

Experiments ran on Dell Precision Workstations with 3GHz Intel Pentium 4 processors, 2GB of RAM and Intel PRO/1000 Gigabit network cards. Hosts were connected through a 100Mbps D-Link Ethernet switch.

We evaluated Vigilante with real worms: Slammer, Blaster and CodeRed. Experiments with CodeRed and Blaster ran on Windows 2000 Server and experiments with Slammer ran on Windows XP with SQL Server 2000. These worms attacked popular services and had a high impact on the Internet.

Slammer infected approximately 75,000 Microsoft SQL Servers. So far, it was the fastest computer worm in history [MPS+03]. During its outbreak, the number of infected machines doubled every 8.5 seconds. Slammer's exploit uses a UDP packet with the first byte set to 0x04 followed by a 375 byte string with the worm code. While copying the string, SQL overwrites a return address in the stack.

CodeRed infected approximately 360,000 Microsoft IIS web servers. It spread much slower than Slammer, taking approximately 37 minutes to double the infected population. CodeRed's exploit sends a "GET /default.ida?" request followed by 224 'X' characters, the URL encoding of 22 Unicode characters (with the form “%uHHHH” where H is an hexadecimal digit), “HTTP/1.0”, headers and an entity body with the worm code. While processing the request, IIS overwrites the address of an exception handler with a value derived from the ASCII encoding of the Unicode characters. The worm gains control by triggering an exception in a C runtime function and it immediately transfers control to the main worm code that is stored in the heap.

Blaster infected the RPC service on Microsoft Windows machines. We conservatively estimate that it infected 500,000 hosts and that its spread rate was similar to CodeRed’s. Blaster is a two-message attack: the first message is a DCERPC bind request and the second is a DCERPC DCOM object activation request. The second message has a field that contains a network path starting with ‘\\’. While copying this field to a buffer and searching for a terminating ‘\’, the RPC service overwrites a return address in the stack.

Additionally, we used the Windows MetaFile (WMF) vulnerability of the Internet Explorer Web browser[1] in our tests. The vulnerability allows an attacker to execute arbitrary code when a user views an image. Windows metafiles contain pictures represented as sequences of calls to the Windows Graphical Device Interface (GDI) library. One of these calls allows the attacker to specify the address of a function that is later called by the GDI. Using this call, the attacker can specify an address corresponding to code inside an attack image file. This vulnerability is not strictly exploitable by a worm, since some user action is still required. However, the attack is serious, since a computer can be immediately infected when a user views an image. It is also interesting in another way: it was exploited on the Internet before it was known by Microsoft.

6.2 Detection

We tested the dynamic data-flow analysis detector on the set of real worm attacks described above, and on a broad range of synthetic attacks. Table 6.1 shows the results for attacks that exploited the network services described above; all the attacks were detected.

[1]Several applications that open image files are affected by this vulnerability.

Service	Attack	Detected?
SQL Server	Slammer attack	yes
Internet Information Server	Code Red attack	yes
Windows RPC Service	Blaster attack	yes
Internet Explorer	Windows Metafile vulnerability	yes

Table 6.1: Real attacks detected by dynamic data-flow analysis.

The synthetic attacks were based on a testbed of 18 buffer overflow attacks described in [WK03]. Each attack is based on a different combination of technique, location and attack target. The testbed uses two techniques, two types of location and four attack targets:

Techniques. The first technique simply overflows a buffer until the attack target is overwritten. The second technique overflows a buffer until a pointer is overwritten, and a uses a subsequent assignment through the pointer to overwrite the attack target.

Locations. The attacks use two types of location for the overflowed buffer: the stack, and the data segment.

Attack Targets. The attacks use four different control data structures as targets: the return address on the stack, the old base pointer on the stack, function pointers and longjmp buffers. The last two can be either variables or function parameters.

Table 6.2 shows the results for the synthetic attacks. All the attacks were detected. It is worth pointing out that dynamic data-flow analysis is able to detect the attacks without using any specific knowledge about the control data structures used by the program. By comparison, the coverage of several tools that protect specific control data structures was tested with the same attacks and the best tools only detected 50 percent of the attacks [WK03]. Even if all the techniques used by the tools tested in [WK03] were combined, a third of the attacks would not be detected.

We also measured the performance overhead introduced by the dynamic data-flow analysis detector with SQL Server, the IIS web server, and the Windows RPC service. For each vulnerable service we measured the average response time of one hundred requests. For SQL Server the requests were generated with transactions

Attack	Target data structure	Detected?
Direct overwrite on stack	Parameter function pointer	yes
	Parameter longjmp buffer	yes
	Return address	yes
	Old base pointer	yes
	Function pointer	yes
	Longjmp buffer	yes
Direct overwrite on data segment	Function pointer	yes
	Longjmp buffer	yes
Overwrite through stack pointer	Parameter function pointer	yes
	Parameter longjmp buffer	yes
	Return address	yes
	Old base pointer	yes
	Function pointer	yes
	Longjmp buffer	yes
Overwrite through data segment pointer	Return address	yes
	Old base pointer	yes
	Function pointer	yes
	Longjmp buffer	yes

Table 6.2: Synthetic attacks detected by dynamic data-flow analysis.

from the TPC-C benchmark [TPC99]. To measure the worst case scenario for detector overhead, we used empty implementations for the TPC-C stored procedures; therefore the requests were CPU bound. For IIS we used requests from the SpecWeb99 [SPE] benchmark. To measure a worst case scenario, IIS returned 512 bytes from main memory in response to every request. For the Microsoft Windows RPC service, we generated a custom workload using requests to lookup an RPC interface; these requests are also CPU bound. Figure 6.1 shows the overhead for each of the experiments for the three vulnerable services. The overhead is large in all cases: the response time increases by a factor of 51 for SQL, 38 for the RPC service, and 50 for IIS. Therefore, it is not appropriate to run our implementation of the dynamic data-flow analysis detector on production services. The largest contributors to the overhead are the Nirvana re-writing mechanism and the disassembler used to decode instructions. Both of these mechanisms can be optimized; for instance, DynamoRIO [BDA00] provides much faster re-writing, and we also plan to optimize the detector by caching decoded instructions. However, as Section 6.3 shows, in spite of their large overhead, these detectors can still generate alerts in times ranging from tens of milliseconds to a few seconds; thus, they can already be used to provide timely detection of unknown worm attacks.

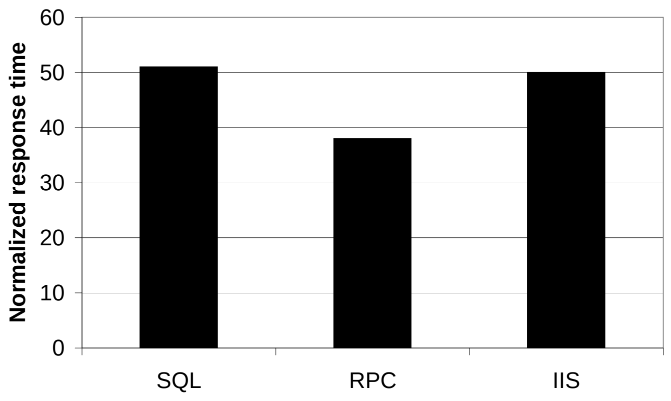


Figure 6.1: Runtime overhead of running the dynamic data-flow analysis detector.

6.3 Alert generation

The next experiment measures the time to generate SCAs with the dynamic data-flow analysis and NX detectors. The time is measured from the moment the last worm message is received till the detector generates an SCA. It does not include the time to verify the SCA before it is distributed and the log contains only the worm messages. One detector uses dynamic data-flow analysis and the other uses a software emulation of non-execute protection on stack and heap pages (*NX*). The detectors generate arbitrary execution control alerts for Slammer and Blaster, and an arbitrary code execution alert for CodeRed.

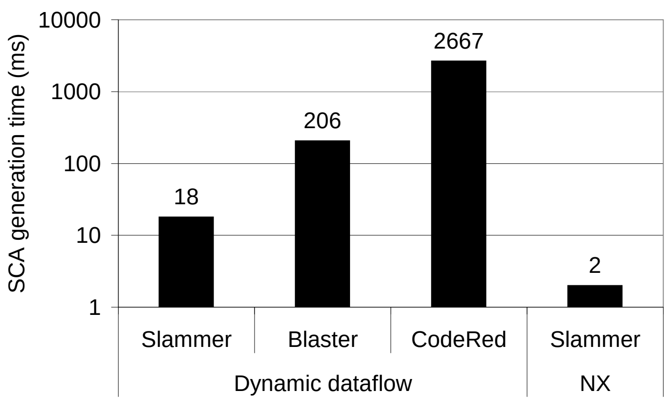


Figure 6.2: SCA generation time in milliseconds for real worms using two detectors.

Figure 6.2 shows average SCA generation times for Slammer, Blaster, and CodeRed with the dynamic data-flow detector and for Slammer using the NX detector. The results are the average of five runs. The standard deviation is 0.5 ms for Slammer, 3.9 ms for Blaster, and 204.7 ms for CodeRed.

Both detectors generate SCAs fast. The NX detector performs best because its instrumentation is less intrusive, but it is less general. For both Slammer and Blaster, the dynamic data-flow detector is able to generate the SCA in under 210 ms and it takes just over 2.6 s for CodeRed. Generation time is higher for CodeRed because the number of instructions executed is larger and Nirvana has to dynamically translate a number of libraries loaded during the worm attack.

Figure 6.3 shows the SCA size in bytes for each worm. The SCAs include a fixed header of 81 bytes that encodes the SCA type, vulnerable service identifica-

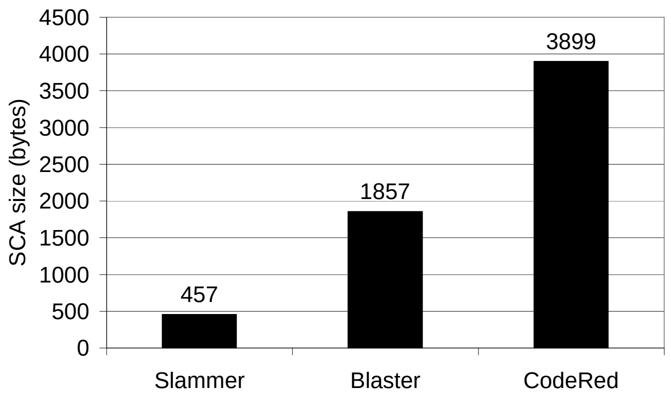


Figure 6.3: SCA sizes in bytes for real worms.

tion and verification information. The size of the SCAs is small and it is mostly determined by the size of the worm probe messages.

6.4 Alert verification

The next experiment measures the time to verify SCAs. SCAs are verified inside a Virtual PC 2004 virtual machine that has all the code needed for verification loaded. The state of this VM is saved to disk before verifying any SCA. After each verification, the VM is destroyed and a new one is created from the state on disk to be ready to verify the next SCA.

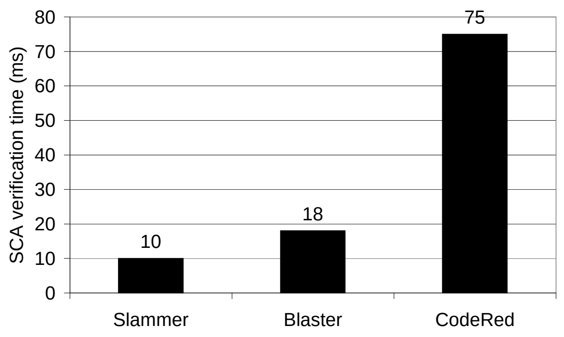


Figure 6.4: SCA verification time in milliseconds for real worms.

Figure 6.4 shows the average time in milliseconds to verify each SCA. The results are the average of five runs. The standard deviation is 0.5 ms for Slammer, 1.5 ms for Blaster, and 6.5 ms for CodeRed.

Verification is fast because it doesn't need to instrument the vulnerable software, and because we keep a VM running that is ready to verify SCAs when they arrive. The overhead to keep the VM running is low: a VM with all vulnerable services used less than 1% of the CPU and consumed approximately 84MB of memory.

We also explored the possibility of starting VMs on demand to verify SCAs. The VM is compressed by the Virtual PC into a 28MB checkpoint. It takes four seconds to start the VM from disk with cold caches, but it takes less than a second to start the VM from a RAM disk. Since this additional delay is problematic when dealing with fast spreading worms, we decided to keep a VM running. Techniques to fork running services [VMC$^+$05; FC03] should enable creation of VMs on demand with low delay.

6.5 Alert distribution

To evaluate the effectiveness of SCA distribution at large scale, we ran simulations with parameters derived from our experiments with the prototype and from published statistics about real worms.

6.5.1 Simulation setup

The simulations ran on a simple packet-level discrete event simulator with a transit-stub topology generated using the topology generator described in [ZCB96]. The topology has 5050 routers arranged hierarchically with 10 transit domains at the top level and an average of 5 routers in each. Each transit router has an average of 10 stub domains attached with an average of 10 routers each. The delay between routers is computed by the topology generator and routing is performed using the routing policy weights of the graph generator. Vigilante hosts are attached to randomly selected stub routers by a LAN link with a delay of 1 ms.

In all the simulations, we use a total population of 500,000 hosts. S randomly selected hosts are assumed *susceptible* to the worm attack because they run the same piece of vulnerable software. A fraction p of the susceptible hosts are randomly chosen to be detectors, while the rest are referred to as *vulnerable* hosts. We evaluate distribution using the secure overlay with super-peers: 1,000 of the 500,000 hosts are randomly selected to be super-peers that form a secure Pastry overlay and each ordinary host connects to two super-peers. Each super-peer is able to verify the SCA and is neither vulnerable nor a detector.

We model worm propagation using the epidemic model described in [Het00] with minor modifications that take detectors into account. Assuming a population of S susceptible hosts, a fraction p of them being detectors, and an average infection rate of β, let I_t be the total number of infected hosts at time t and P_t be the number of distinct susceptible hosts that have been probed by the worm at time t, the worm infection is modelled by the following equations:

$$\frac{dP_t}{dt} = \beta\ I_t(1 - \frac{P_t}{S}) \tag{6.1}$$

$$\frac{dI_t}{dt} = \beta\ I_t(1 - p - \frac{I_t}{S}) \tag{6.2}$$

Starting with k initially infected hosts, whenever a new vulnerable host is infected at time t, our simulator calculates the expected time until a new susceptible host receives a worm probe using Equations (6.1) and (6.2), and randomly picks an unprobed susceptible host as the target of that probe. If the target host is vulnerable, it becomes *infected*. If the target host is a detector, an SCA will be generated and distributed.

To account for the effects of network congestion caused by worm outbreaks, we built a simple model that assumes the percentage of packets delayed and the percentage of packets dropped increase linearly with the number of infected hosts. We computed the parameters for the model using the data gathered during the day of the Slammer outbreak by the RIPE NCC Test Traffic Measurements (TTM) service [GGK+01]. At the time, the TTM service had measurement hosts at 54 sites spread across the world and each host sent a probe to each of the other hosts every 30 seconds.

Since Slammer took approximately 10 minutes to propagate, we computed the peak percentage of packets dropped and delayed by analyzing the data during the 10-minute interval starting at 10 minutes after the Slammer outbreak. We also computed the average increase in packet delay using as the baseline the delays in the 10-minute interval ending at 10 minutes before the outbreak. We observed that about 9.6% of the packets sent were delayed with an average delay increase of 4.6 times, while 15.4% of the packets were dropped. We delay or drop a percentage of packets equal to the above values multiplied by the fraction of infected hosts.

When probed, a detector takes time T_g to generate an SCA and then it broadcasts the SCA. SCA verification takes time T_v. Detectors, vulnerable hosts, and super-peers can verify SCAs but other hosts cannot. Unless otherwise stated, we assume 10 initially infected hosts. Each data point presented is the mean value of 250 runs with an error bar up to the 90^{th} percentile. Each run has different random choices of susceptible hosts, detectors, and initially infected hosts.

We model a DoS attack where each infected host continuously sends fake SCAs to all its neighbours to slow down distribution. We conservatively remove rate control. We assume that the concurrent execution of n instances of SCA verification increases verification time to nT_v seconds.

Finally, we note that while accurately modelling worm outbreaks and countermeasures is still an area of active research [MSVS03; ZGGT03; CGK03; VG05; GGK+06], the worm spreading model above has been shown to describe accurately outbreaks of real worms [MPS+03], and we parameterized the model with measurements from our implementation of Vigilante and with data collected during real outbreaks.

6.5.2 Containment of real worms and beyond

First, we evaluate the effectiveness of Vigilante with Slammer, CodeRed, and Blaster. Table 6.3 lists the parameter settings used for each worm. The infection rates (β) and susceptible population (S) for Slammer and CodeRed are based on observed behaviour reported by Moore et al. [MPS+03]. Blaster was believed to be slower than CodeRed, but with a larger susceptible population. We conservatively set its infection rate to be the same as CodeRed and have the entire population being susceptible. T_g and T_v are set according to the measurements in Sections 6.3 and 6.4.

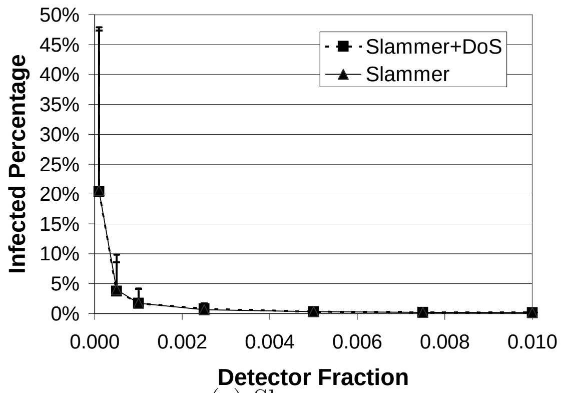
	β	S	T_g (ms)	T_v (ms)
Slammer	0.117	75,000	18	10
CodeRed	0.00045	360,000	2667	75
Blaster	0.00045	500,000	206	18

Table 6.3: Simulation parameters for modelling containment of real worms.

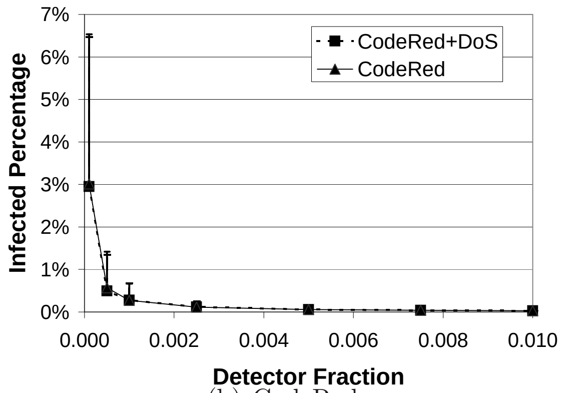
Figure 6.5 shows the infected percentage (i.e., the percentage of vulnerable hosts that are eventually infected by the worm) for the real worms with different fractions (p) of detectors both with and without DoS attacks. The graph shows that a small fraction of detectors ($p = 0.001$) is enough to contain the worm infection to less than 5% of the vulnerable population, even under DoS attacks. The Vigilante overlay is extremely effective at disseminating SCAs: once a detector is probed, it takes approximately 2.5 seconds (about 5 overlay hops) to reach almost all the vulnerable hosts.

SCA verification time (T_v) determines SCA distribution delay, whereas the number of initially infected hosts (k) and infection rate (β) characterize worm propagation. Figure 6.6 shows the impact of T_v, β, and k on the effectiveness of Vigilante, both with and without DoS attacks. Slammer is the fastest propagating real worm. We therefore use Slammer's $\beta = 0.117$ as the base value in subfigure (b), for example, with a worm infection rate of 8β, the number of infected machines doubles approximately every second. Because the initially infected hosts are counted in the infected percentages reported, the baseline in subfigure (c) shows the contribution of the initially infected hosts to the final infected percentage. Unless otherwise specified, the experiments use the default values with p of 0.001, k of 10, T_g of 1 second, T_v of 100 ms, β of 0.117, and S of 75,000.

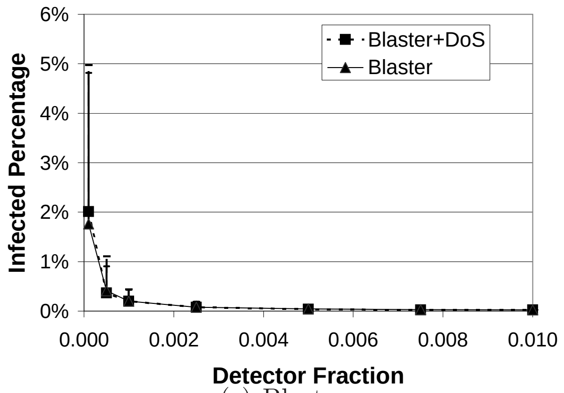
These results show that Vigilante remains effective even with significant increases in SCA verification time, infection rate, or number of initially infected hosts. The effectiveness of Vigilante becomes reduced (and exhibiting significant variations) with SCA verification time of 1000 ms, with infection rate of 8β, or with 10000 initially infected nodes. Do note that those settings are an order of magnitude worse than the worst of real worms.



(a) Slammer

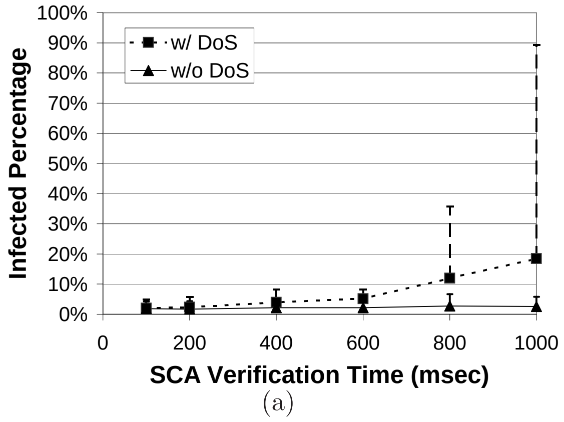


(b) CodeRed

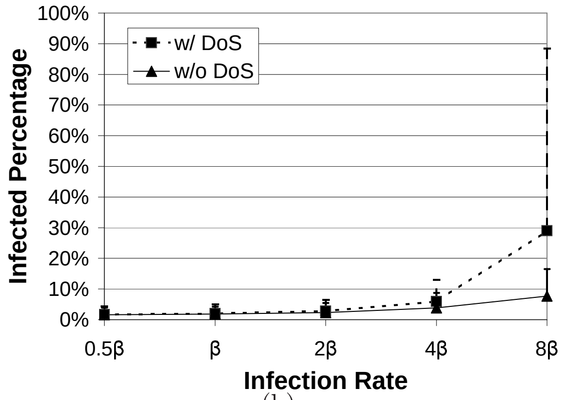


(c) Blaster

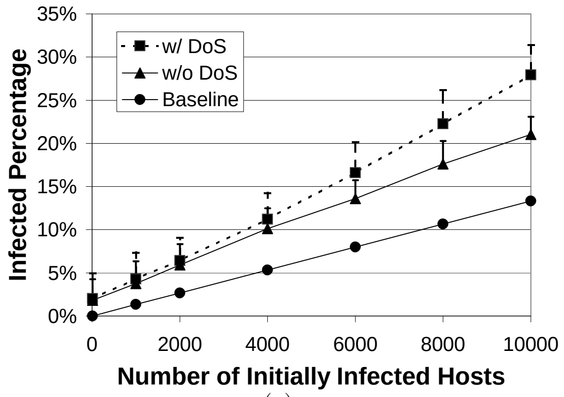
Figure 6.5: Containment of Slammer, CodeRed, and Blaster using parameter settings in Table 6.3, both with and without DoS attacks. Each data point is the mean value with an error bar up to the 90^{th} percentile value.



(a)



(b)



(c)

Figure 6.6: The effect of SCA verification time, infection rate, and number of initially infected hosts, both with and without DoS attacks. Each data point is the mean value with an error bar up to the 90^{th} percentile value.

Not surprisingly, DoS attacks appear more damaging in configurations where Vigilante is less effective because the significance of DoS attacks hinges directly on the number of infected hosts. Also as expected, Vigilante is increasingly vulnerable to DoS attacks as the verification time increases.

Other attacks on the distribution of SCAs have also been analyzed in recent work: [RHR06] analyzed the possibility of using the information in the SCAs to generate new worms. Such attacks have a limited impact, because the generated worms compete with a worm that is already spreading.

6.6 Protection

The next set of experiments evaluates the overheads associated with filters and their effectiveness.

6.6.1 Filter generation

The first experiment measures the time to generate a filter from an SCA that has already been verified. Figure 6.7 shows the time in milliseconds to generate both the specific and general filters for the three worms. The results are the average of five runs. The standard deviation was 0.7 ms for Slammer, 5.1 ms for Blaster, and 205.3 ms for CodeRed. In all cases, filter generation is fast. Filter generation for CodeRed is more expensive because the number of instructions analyzed is larger and the binary re-writing tool needs to dynamically translate code for a number of libraries that are loaded on demand.

The generated filters are also effective. In all cases, the specific filters block the attack, have no false positives, and also block many polymorphic variations of the attack. We describe the general filters in more detail because they determine the false negative rate.

The general filter for Slammer checks that the first byte is 0x4 and that the following bytes are non-zero (up to the byte offset of the value that would overwrite the return address in the stack). This filter is optimal: it captures all polymorphic variations of the attack with no false positives. The filter's code sequence is not optimized: it corresponds to a stack-based evaluation of the filter

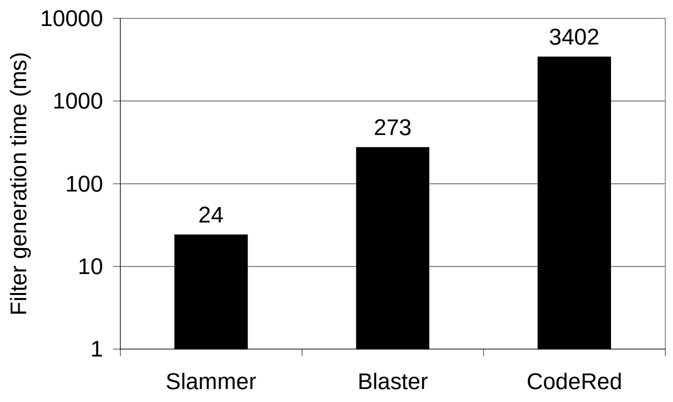


Figure 6.7: Filter generation time for real worms.

condition. For example in Slammer, the condition that the first byte is equal to 0x04 is computed by the code in Figure 6.8. There are a number of obvious optimizations, but the performance of the filter is good even without them.

The general filter for Blaster checks that there are two consecutive backslash (‘\’) Unicode characters at the required positions, followed by Unicode characters different from ‘\’ up to the position of the value that will overwrite the return address in the stack. This filter catches all polymorphic variations in the worm code and some variations in other parts of the message.

The general filter for CodeRed checks that the first 4 bytes form the string “GET ”, and that bytes from offset 0x11 to offset 0xF0 are ASCII characters and that they are different from ‘+’ and ‘%’. The filter also checks that “%u” strings are used in the same positions where the attack used them and that the characters following those strings are ASCII representations of hex digits. This filter catches polymorphic variations on the worm code and insertion of HTTP headers in the attack message.

These results show that dynamic control and data flow analysis is a promising approach to filter generation. While the general filter for Slammer is perfect, the general filters for Blaster and CodeRed have some limitations. For Blaster, it is possible that other successful attacks could be mounted by using the string starting with ‘\’‘\’ at a different position in the attack message. The CodeRed filter also does not tolerate shifting or insertion of ‘+’ or ‘%’ where the worm used ‘X’ characters. We plan to improve the general filters in the future. In our

```
xor eax,eax                    ;clear the eax register
mov al,byte ptr [esi + 0x0]    ;move first byte into al
push eax                       ;push the first byte into the stack
push 0x02
pop ebx
pop eax
sub eax,ebx                    ;subtract 2 from first byte
push eax
pop eax
mov ebx,0x02
cmp eax,ebx                    ;compare with 2
jne do_not_drop                ;exit the filter without a match if not equal
```

Figure 6.8: IA-32 code for a condition of Slammer's filter.

current implementation, filters may also be evaded with packet fragmentation. We plan to address this limitation by implementing well known countermeasures for this evasion technique [PN98].

6.6.2 Overhead of deployed filters

We also measured the performance overhead introduced by deployed filters. Filters were deployed by binary re-writing the vulnerable services. We used Detours [HB99] to intercept calls to the socket interface and install the filters immediately above functions that receive data.

We ran three experiments for each vulnerable service and measured the overhead with a sampling profiler. The first experiment (*intercepted*) ran the service with just the socket interface being intercepted. The second experiment (*intercepted + filter*) ran the service with the socket interface being intercepted and invoking the appropriate general and specific filters. The third experiment (*intercepted + filter + attack*) stressed the filter code by sending worm probes to the service at a rate of 10 per second (which is three orders of magnitude larger than the rate induced by Slammer). For every experiment, we increased the service load until it reached 100% CPU usage, as described below. Figure 6.9 shows the overhead for each of the experiments for the three vulnerable services. The results are the average of five runs. The overhead is very low in all cases.

SQL For Slammer the vulnerable service is SQL Server. We generated load using the TPC-C benchmark [TPC99] with 170 simulated clients running on two separate hosts. Clients were configured with zero think time. To measure the worst case scenario for the filter overhead, the number of requests serviced per unit time was maximized by using empty implementations for the TPC-C stored procedures. Figure 6.9 shows that the CPU consumed by the interception is just 0.16%. When then Slammer filters are installed, the overhead remains the same because Slammer exploits a vulnerability in a management interface running on UDP port 1434. This is not the same communication endpoint that SQL uses to listen for client transactions. Therefore, the requests issued by the TPC-C clients follow a different code path and the impact of running the filter is negligible. With worm probes, the overhead rises to only 0.2%.

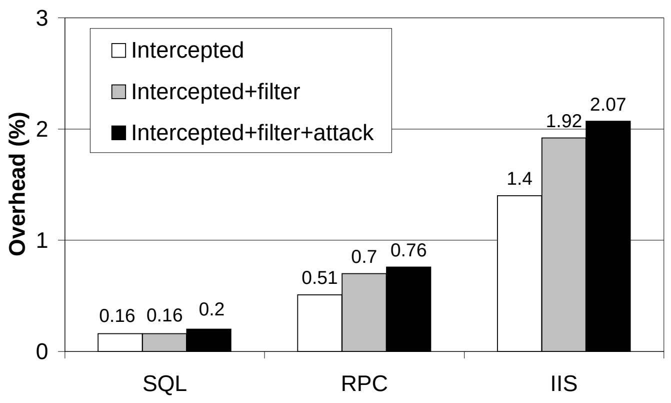


Figure 6.9: CPU overhead of network traffic interception and filter deployment.

RPC For Blaster the vulnerable service is Microsoft Windows RPC service. We generated a custom workload using requests to lookup and register an RPC interface. We loaded the RPC service using 3 client hosts that lookup the RPC interface and 1 local client that registers the interface. Figure 6.9 shows the CPU consumed by interception is only 0.51%, and it rises to 0.7% when the filters are invoked. When running with 10 Blaster probes per second the overhead was 0.76%. Unlike Slammer, the filters are on the normal execution path and are used by requests to lookup the interface.

IIS For CodeRed the vulnerable service is Microsoft IIS Server. We generated a workload using the requests from the SpecWeb99 [SPE] benchmark with clients running on two separate hosts. To measure a worst case scenario for filter overhead, we installed an IIS extension that returns 512 bytes from main memory in response to every request. Figure 6.9 shows that the CPU consumed by the interception is 1.4%. The majority of this CPU overhead is attributable to matching I/O operation handles to discover where data is written when asynchronous I/O operations complete. When the CodeRed filters are invoked the overhead increases to 1.92%. These filters are on the normal execution path and are invoked for every packet. Finally, adding the 10 CodeRed probes per second, the overhead rises to 2.07%.

6.7 End-to-End experiments

The final set of experiments measures Vigilante's worm containment process end-to-end in a five-host Vigilante network. The hosts were configured in a chain representing a path from a detector to a vulnerable host in the SCA distribution overlay with three super-peers. They were connected by a LAN. The first host was a detector running a dynamic data-flow analysis engine. Once the detector generated an SCA it was propagated through three super-peers to a host running the vulnerable service. This provides approximately the same number of hops as the simulations in Section 6.5.

We measured the time in milliseconds from the moment the worm probe reached the detector till the moment when the vulnerable host verified the SCA. This time is critical for protection. After successful verification, the vulnerable host can suspend execution of the vulnerable service during filter generation. We ran the experiment for the three worms: using SQL Server with Slammer, the Windows RPC Service with Blaster, and IIS with CodeRed. The time was 79 ms for Slammer, 305 ms for Blaster, and 3044 ms for CodeRed. The results are the average of five runs. The standard deviation is 12.2 ms for Slammer, 9.0 ms for Blaster and 202.0 ms for CodeRed. These values are close to those obtained by adding the SCA generation time to five SCA verifications, as expected.

The vulnerable host deployed the filter after it was generated, which does not require re-starting the vulnerable service. To achieve hot installation of the filters, the functions that intercept the socket API check for availability of filters on a shared memory section. After filter generation, the filter code is copied to the vulnerable process through the shared memory section. Filter deployment is fast: in all cases filters were deployed in less than 400 microseconds.

Chapter 7

Related work

Previously proposed techniques to mitigate worm attacks can be divided into network-based and host-based mechanisms. Network-based mechanisms exclusively analyze network traffic, while host-based systems use information available at the end-hosts. This chapter discusses previous proposals in each of these areas.

7.1 Network-based mechanisms

Detection in network-based systems is based on defining a model of normal traffic and identifying deviations from that model. Protection in these systems consists of blocking suspicious traffic. Traffic can be considered suspicious for several reasons: it may come from outside an enterprise network perimeter; it may come from machines thought to be infected; it may match a signature generated from previously observed attacks; or it may contain suspicious data (e.g. data that looks like executable code). All network-based systems that we are aware of are based on heuristics and can have both false positives and false negatives. Furthermore, it seems difficult to completely remove false positives and false negatives from these systems, because the root cause for worm attacks, vulnerable programs, is not visible at the network level.

7.1.1 Firewalls

Firewalls [CBR03] are one of the most successful network-based protection mechanisms. Enterprise firewalls define a boundary between enterprise networks and the Internet. Only certain types of network interactions are allowed across the firewall boundary. For instance, incoming connections are usually disallowed. Firewalls are effective at blocking many attacks, but they are a brittle boundary. Worms can bypass them using web browser vulnerabilities or email-based attacks, because firewalls typically allow this type of traffic [CER01]. Worms can also exploit virtual private network connections and infected laptop computers to penetrate enterprise networks. After infecting one computer inside the enterprise network, the worm can spread internally unhampered by the firewall. Thus, while firewalls make it hard for the worm to directly send attack messages from the Internet to computers on enterprise networks, they do not provide a general solution for containment.

Personal firewalls, i.e. firewalls that run on personal computers, are also widely deployed. They are usually more permissive than enterprise firewalls, and therefore less effective at blocking attacks. Personal firewalls provide an effective mechanism to deploy traffic filters generated with the blacklisting and content filtering approaches discussed next.

7.1.2 Address blacklisting

Several systems are based on the idea of blocking network traffic from infected computers, thus preventing them from infecting other computers. Early proposals identified infected computers by analyzing host connectivity graphs [SCCD^{+}96]. The heuristics used by the GrIDS system generated 1 to 2 false positives a day; it is unclear how many false positives would be generated by current traffic. More recently, several systems proposed identifying infected machines by detecting scanning behaviour. Mirage networks [Mir06] and Forescout [For06] mark machines as infected if they send messages to unallocated (dark) IP addresses. Worms can avoid this type of detectors by not using dark IP addresses. The systems in [GEB02; WKO05] consider machines infected if they use IP addresses without first resolving the corresponding DNS [Moc87] names. These systems can generate false positives that need to be handled with whitelisting. It seems they can also be evaded if worms coordinate to fake DNS traffic. For instance, a worm instance can generate DNS queries that are answered by another worm instance, by supplying the appropriate IP address for the next scan target.

Several systems detect scanning by observing that worms generate many failed network transmissions [TK02b; JPBB04; SJB04; WSP04], because they try to contact unreachable addresses. Jung et al. [JPBB04] proposed Threshold Random Walk (TRW): an algorithm that can be parameterized with models of good traffic and attack traffic, and detects infection by analyzing the rate of successful to failed connections. Weaver et al. [WSP04] proposed a simplification of TRW that uses a threshold on an estimate of the difference between the number of failed connections and the number of successful connections. Bro [Pax99] uses a configurable threshold on the number of failed connections. Snort [Roe99] and Network Security Monitor [HDK+90] do not look at failed connections; instead they monitor the rate at which unique destination addresses are contacted. If computers exceed a threshold of new addresses contacted in a given interval, they can be flagged as infected. Finally, SPICE [SHM02] is an algorithm to detect very slow scans of enterprise networks by correlating anomalous events; the algorithm gathers information over long time periods(days) and is expensive to run. Therefore it is not well adapted to the detection of fast spreading worms.

Staniford [Sta04] and Ganesh [GGK+06] analyze the conditions under which scanning detection and subsequent blacklisting can provide containment. Staniford [Sta04] discusses the importance of an "epidemic threshold" for these systems: if on average an infected machine can find more than one victim before being blacklisted, the number of infected machines will still grow exponentially. Weaver et al. argue [WESP04] that scanning detection and suppression would need to be deployed in every local area network (LAN), in special hardware devices, for the system to provide containment.

These systems also cannot contain worms that have normal traffic patterns, for example, topological worms that exploit information about hosts in infected machines to propagate, thus avoiding scanning. False positives are another problem for these systems, because several normal network services exhibit scanning-like behaviour [Jun06]. A related problem is malicious false positives, for example, an attacker can perform scanning with a fake source address to block traffic from that address.

7.1.3 Throttling connections

A variant of blacklisting is throttling: limiting the resources used by infected machines, without blocking all traffic from those machines. Williamson [Wil02] proposed limiting the rate of connections to new addresses. This approach limits the impact of false positives, by allowing the machines to continue active, albeit with degraded performance. On the other hand it only slows the spread of worms, without providing containment.

7.1.4 Content filtering

Another approach to network-based worm containment is to generate a set of content signatures for worm attack messages, and to drop messages that match the signatures. Interest in this approach increased after Moore et al. [MSVS03] showed it is superior to blacklisting, if content signatures can be generated quickly. The intuition for this is simple: systems based on blacklisting need to continuously discover and blacklist the addresses of the infected machines very soon after they become infected, while content filtering systems can block all attack traffic by generating a signature only once.

Worm signatures have traditionally been generated by humans but there are several proposals to generate signatures automatically. Kephart et al. [KA94] proposed, in the context of viruses, the first algorithm to generate signatures automatically. Their system generates byte string signatures by luring viruses into infecting decoy programs, and creating candidate signatures by finding common substrings in several instances of infected programs[1]. The candidate signatures are then filtered to minimize the probability of false positives.

More recently, Honeycomb [KC03] proposed generating byte string signatures from the traffic observed at honeypots. Honeycomb assumes all traffic received by

[1]Strictly speaking, this system uses host-level information, but we include it here since it is similar to subsequent network-based systems that generate signatures by finding common substrings in network traffic.

honeypots is suspicious. Signatures are generated by finding the longest common substring in two network connections. The system can generate false positives if legitimate traffic reaches the honeypot. Malicious false positives are also a problem, since an attacker can send traffic to the honeypot in order to generate a signature. Honeycomb can also have false negatives. It uses a configurable minimum length for its signatures, to avoid false positives, but this will allow polymorphic worms to spread undetected. Polymorphic worms can have little invariant content across attack messages, thereby making it difficult to match them with byte strings.

Autograph [KK04] also generates byte string signatures automatically. Rather than relying on honeypots, Autograph identifies suspicious network flows at the firewall boundary. It stores the address of each unsuccessful inbound TCP connection, assuming the computer generating such connection requests is scanning for vulnerable machines. When a configurable number of such attempts are recorded, Autograph marks the source IP address as infected. All subsequent connections involving IP addresses marked as infected are inserted into a pool of suspicious network flows. Periodically, Autograph selects the most common byte strings in the suspicious flows as worm signatures. To limit the amount of false positives, Autograph can be configured with a list of disallowed signatures; the authors suggest a training period during which an administrator runs the system and gradually accumulates the list of disallowed signatures. The system is also configured with a minimum signature size, which can result in false negatives, especially with polymorphic worms.

Earlybird [SEVS04] is based on the observation that it is rare to see the same byte strings within packets sent from many sources to many destinations. Unlike Autograph, Earlybird doesn't require an initial step that identifies suspicious network flows based on scanning activity. Earlybird generates a worm signature, when a byte string is seen in more than a threshold number of packets and it is sent/received to/from more than a threshold number of different IP addresses. Earlybird uses efficient algorithms to approximate content prevalence and address dispersion; therefore, it scales to high-speed network links. To avoid false positives, Earlybird uses whitelists and minimum signature sizes. As with Honeycomb and Autograph, malicious false positives are a concern and polymorphic worms are likely to escape containment.

PayL [WCS05] is based on the idea of analyzing byte frequency distributions in normal traffic, and considering messages with anomalous distributions as suspect. PayL triggers a signature generation procedure if outgoing messages are similar to suspect incoming messages. PayL signatures are byte strings which are shared by incoming and outgoing suspect messages. PayL can generate false positives and recent work [FSP+06] showed that it can be evaded.

Polygraph [NKS05] argued that single byte string signatures cannot block polymorphic worms. In an effort to generate signatures that match polymorphic worms, Polygraph generates signatures that are multiple disjoint byte strings, instead of a single byte string. Polygraph relies on a preliminary step that classifies network flows as suspicious or innocuous. Tokens are identified as repeated byte strings across the suspicious network flows. A subsequent step groups tokens into signatures. Polygraph proposes three types of matching with these signatures: matching all the byte strings in a signature, matching the byte strings in order, or assigning a numeric *score* to each byte string and base matching in an overall numeric threshold. Their evaluation shows that none of these types of signature is superior to the others for every worm. All of them can have false positives and false negatives. A recent evaluation [PDL+06] shows that attacks that generate fake anomalous network flows can prevent Polygraph from reliably generating useful signatures.

PADS [TC05] generates signatures that are a sequence of byte frequency distributions. The authors show that PADS works for some cases, but it is unclear if a polymorphic worm cannot generate arbitrary byte frequency distributions for most bytes in the attack messages. Malicious false positives are also a problem for PADS; it uses a configuration with two honeypots to try to remove any non-worm traffic from the signature generation procedure, but the worm can still generate bogus traffic after infecting a machine.

Nemean [YGBJ05] uses protocol-specific information to generate signatures that are regular expressions and may include session-level context, but it requires some manual steps and also cannot cope with pollution of the network data that is used as input to the signature generation process.

Finally, another technique to filter attack messages is to identify executable code in network messages. Toth and Kruegel [TK02a] proposed using binary disassembly over a network flow and dropping messages whenever a long sequence of valid instructions is found. An instruction is considered valid if it can be decoded by the processor and if all the memory operands of the instruction reference memory locations that can be accessed. Strictly speaking, this mechanism requires host-based information, since checking if the memory locations can be accessed requires having access to the address space of the process running the target program. However, this information can easily be approximated (e.g. certain memory regions are always reserved for the operating system and can never be accessed by applications) and subsequent systems removed this requirement [CvdB05; KKM+05a; WPLZ06]. Their system assumes attack messages will have a relatively long region with instructions that have no effect (sometimes called a NOP *sledge* [TK02a]), because this is common technique used by worms to deal with small variations on the location where attack messages are stored in the virtual address space of target processes. This technique can be defeated by inserting noise (e.g. branch instructions, illegal instructions, etc) in the sledge. To deal with this type of attacks, several systems [CvdB05; KKM+05a; WPLZ06] proposed using static analysis techniques on the disassembled network flow. These systems identify executable code in the network flow more reliably, at some performance cost.

The techniques that identify code in messages are more resilient to attack mutations, because they do not use fixed byte strings as signatures. They may still have false negatives because they look for code sequences of some minimum length (e.g. 15 instructions [WPLZ06]) and worms can use very short code sequences to encode/decode the bulk of the attack payload. Another source of false negatives is worm attacks that succeed without injecting new executable code into their targets. Even for injected code, the code may be encoded in the protocol messages [rix01]; for instance the systems in [TK02a; WPLZ06] use protocol specific information to decode the network messages, before trying to find executable code.

7.2 Host-based mechanisms

Host-based mechanisms either statically analyze programs, or dynamically analyze the execution of programs. Some host-based mechanisms try to remove or avoid all defects that might be exploited by worms, while other systems detect attacks only when worms exploit defects at runtime. The latter often require additional survivability mechanisms, since detection is usually not enough to keep programs running while they are being attacked. This section reviews work in all of these areas.

7.2.1 Avoid/Remove defects

Type safe languages [Car04; Mad06] can avoid many of the defects that can be exploited by worms. However, these languages force the programmer to relinquish some of the flexibility and speed available in languages like assembly or C; thus, they have not been adopted by some programmers. Many of these languages include facilities to link with unsafe modules, and often their runtimes are written in unsafe languages. This has made them vulnerable to attacks [Sec02]. Finally, there is a very large body of code written in unsafe languages; the effort of porting this code to different languages is large and difficult to justify economically. Languages like CCured [NMW02] and Cyclone [JMG$^+$02] try to facilitate the evolution of code written in C to memory-safe dialects. The disadvantage of these approaches is that the effort to port existing C code to these dialects is non-trivial and they require significant changes to the C runtime, for example, CCured replaces malloc and free by a garbage collector.

Another approach to remove defects is to statically analyze the source code of programs, looking for specific classes of defects. SELECT [BEL75] and Lint [Joh84] were some of the early tools in this space. More recently, several tools [BPS00; YTEM04; XA05] have been used to find defects in large programs. Some tools have been specifically designed to find security vulnerabilities [WFBA00; EL02; JW04; STFW01; LE01; ADLL05; LL05].

Most of these tools can generate false positives, i.e. they report defects which are not real. One reason for this is that their results may be based on control-flow paths that are infeasible at runtime, but they cannot determine this statically.

They also often have limits on the length of execution paths they explore, to be able to scale to large programs, but this causes false negatives. Unsound handling of pointer aliasing may also create false negatives. Finally, they may also have false negatives because they usually look for known classes of defects. Hence, they cannot find previously unknown types of defects, although there has been some work on describing defects generally as deviant behaviour [ECH+01].

7.2.2 Detect/Prevent exploits

Since static tools can have false positives and they have not been able to remove all defects from software, runtime mechanisms have been developed to detect and stop attacks at runtime. These systems are based on the idea of detecting or preventing exploits, rather than removing defects.

One of the first host-based techniques to detect attacks is to identify anomalous patterns of system calls [FHSL96]. Wagner et al. [WS02] showed that *mimicry attacks* can elude this type of detection, and Kruegel et al. [KKM+05b] showed how to automate these attacks, even for recent improvements on the original technique [FKF+03; GJM04; SBDB01].

Other early systems protected specific control data structures, such as return addresses. StackGuard [CPM+98] proposed writing a *canary* value between the local variables and the return address on a stack frame, and checking that the canary value is intact, before using the saved return address. This detects attacks that overflow buffers on the stack, because the overflow overwrites the canary value on the way to overwriting the return address. StackShield [Ven01], RAD [CH01], and Libverify [BST00] proposed keeping a copies of return addresses separate from the normal stack. This allows them to detect overwrites of return addresses by comparing the saved values with the values on the normal stack. They can also recover the original return addresses. Libsafe [BST00] provided implementations of C library functions that do additional bound checks to avoid overwriting return addresses. FormatGuard [CBB+01] provides safe implementations of C library functions that use format strings. PointGuard [CBJW03] proposed protecting pointers by encrypting them in memory and decrypting them when they are loaded into registers. While effective at protecting some attack targets, these approaches can be bypassed [WK03; BK00].

More recently, DIRA proposed protecting all control data structures [SC05] by keeping a separate copy of these data structures and checking their integrity at control-flow transfers. The copies are protected by storing them between guard (read-only) memory pages. Such protection can be bypassed by corrupting pointers, and using assignments though the corrupted pointers to directly change the stored copies, without writing over the guard pages [WK03; CH01].

Backwards-compatible bounds checking for C [JK97] detects bounds errors in C programs. It instruments pointer arithmetic to ensure that the result and original pointers point to the same object. To find the target object of a pointer, it uses a splay tree that keeps track of the base address and size of heap, stack, and global objects. A pointer can be dereferenced provided it points to a valid object in the splay tree. CRED [RL04] is similar but provides support for some common uses of out-of-bounds pointers in existing C programs. These systems may have false negatives, since they do not prevent all bounds violations. For example, they cannot prevent attacks that exploit format string vulnerabilities or that overwrite data using a pointer to a dead object whose memory was reused. Additionally, they have high overhead because of accesses to the splay tree; for instance, the scheme in [JK97] can cause up to a 30X slowdown in applications. The overhead may be controlled by applying the checks only to specific types of data(e.g. strings) [RL04], yielding a slowdown of up to 2.3X, but this increases the number of false negatives.

Program shepherding [KBA02] introduced a general mechanism to ensure that a program does not deviate from its control-flow graph. They compute a control-flow graph for a program statically, and they use a dynamic binary re-writer [BDA00] to monitor the program's execution and ensure that every control-flow transition is allowed by the control-flow graph. Control-Flow Integrity [ABEL05] checks that control-flow transitions follow the computed control-flow graph with inlined checks based on a static binary re-writer.

Program shepherding has less overhead than current implementations of dynamic data-flow analysis, but it has several limitations. Program shepherding cannot detect attacks that succeed without changing the control-flow of the target programs [CXS+05]. Dynamic data-flow analysis can detect some of these attacks, for example, attacks that overwrite arguments of system calls with data received from the network. Also, program shepherding cannot be used on programs for which it is not feasible to compute a control-flow graph statically. Dynamic data-flow analysis works even with self-modifying code. Finally, program shepherding requires access to source code, while dynamic data-flow analysis works on unmodified binaries.

Concurrently with the publication of the dynamic data-flow analysis algorithm presented here [CCCR04; CCC+05], three systems [SLD04; CC04; NS05] have proposed similar mechanisms for detection, that do not require access to source code. The idea of tracking input data and preventing unsafe uses of that data, can be traced back to Perl taint mode [Per06], and Chow at al. [CPG+04] proposed tracking the lifetime of sensitive information, such as passwords, through memory and CPU registers. More recently, Suh et al. [SLD04] proposed a hardware design that tracks the flow of data from I/O operations. Their design tags each byte of memory with a dirty bit, but they also include multi-granularity tags, to optimize storage and bandwidth overhead. Besides tracking direct copies of input data, their system can also track three other forms of dependency: when a dirty value is used in arithmetic or logic instructions, the result of the operation may be marked dirty; when a dirty value is used to specify an address in an instruction that loads data from memory, the loaded value may be marked dirty; when an instruction that stores data in memory uses a dirty value to specify the address of the store, the stored value may be marked dirty. Since tracking all of these dependencies may generate false positives, the system allows users to specify a per-application security policy, describing which I/O flows should be tracked, which dependencies should be tracked, and which uses of dirty data should generate security traps. They also include some heuristics to reduce false positives; for instance, they identify common code patterns that are safe, but would normally be trapped as attacks (e.g. using a dirty value to index a jump table, after appropriate bounds checking is performed); these heuristics may lead to false negatives. They do not detect use of dirty data in system function calls; we believe this is an important avenue for attacks.

Minos [CC04] is a hardware microarchitecture that implements Biba's low-water-mark policy [Bib77]. In Minos, every 32 bit word is tagged with an additional bit. Since Intel CPUs can address memory at byte granularity, tagging 32 bit words leads to imprecision, which may cause false positives (e.g. a word is marked dirty when only one of its bytes is dirty; the clean bytes may be moved to another location, causing it to be tagged as dirty, when it is in fact clean). By contrast, Vigilante's dynamic data-flow analysis tags each individual memory byte with an additional bit. In Minos, when dirty data is combined with clean data using arithmetic and logic instructions, the resulting data is marked dirty; this increases coverage at the cost of a possible increase in false positives. While Minos does not propagate dirtiness when stores or loads use a dirty value to specify the address, it marks as dirty values resulting from 8 and 16 bit immediate loads; this increases coverage when network data is used in some addressing operations (e.g., table lookups for character translation), but it also increases false positives. To increase coverage, Minos can track network data across disk operations, but this requires changes to the operating system. Vigilante does not track the flow of data when it leaves the address space of a process. Minos only detects attacks that hijack control-flow by overwriting control data structures. Vigilante also detects attacks that corrupt non-control-data used in system calls, and attacks that redirect execution to dirty memory regions, without corrupting control data structures.

TaintCheck [NS05] tracks input data by instrumenting binaries using Valgrind [NS03]. TaintCheck tags each byte of dirty memory with a 32 bit pointer to a data structure that records the system call through which the data was received into the address space of the process, a copy of the stack at the time when the data was received, and a copy of the data. TaintCheck propagates dirtiness when executing data movement and arithmetic operations. As Minos, it does not check if execution is redirected to a dirty memory region, which is important to catch some attacks (it only checks if the value loaded into the program counter is dirty). As Vigilante, Taincheck also checks the dirtiness of arguments to security sensitive functions. TaintCheck proposes using a training phase to deal with false positives: locations where false positives were observed can be recorded to avoid raising security traps there.

The work in [CXN+05] evaluated a security policy that generates security traps when memory writes use dirty pointers. This policy had been proposed in [SLD04], but not evaluated in the context of non-control data attacks. This technique can catch some attacks that do not change the control-flow of programs, but it also increases the likelihood of false positives. Crandall et al. [CC04] discusses the possibility of checking the integrity of addresses used in 32 bit loads and stores. They conclude that this approach is infeasible, because it would generate too many false positives, if dirtiness is also propagated by arithmetic and logic instructions. Vigilante's procedure to verify SCAs provides an effective way to deal with this type of false positives. If a detector generates an alert that cannot be verified, it is simply discarded.

Since its original publication [CCCR04; CCC+05], the dynamic data-flow analysis algorithm has also be used by several systems. [HFC+06] proposed an implementation based on the Xen [BDF+03] virtual machine monitor that automatically transitions from emulation to direct CPU execution, when none of the CPU's registers are dirty. Argos [PSB06] uses an implementation based on QEMU [QEM06] to detect attacks on full operating system and application code.

Another host-based approach to thwart attacks is randomization. Several forms of achieving diversity through randomization were initially discussed in [FSA97]. Randomizing the memory layout of processes was originally implemented by the PaX [PAX01] project. Randomizing the location of the stack, heap and code makes it difficult for the attacker to gain control of the target program: even if the attacker can force the program to load an arbitrary value into the program counter, it's still difficult to know which value to supply (since the attacker doesn't know, for instance, where the attack messages are in the target's address space). Recent projects proposed improvements on this technique [BDS03; BSD05; XKI03]. Several attacks against address randomization have been proposed [Dur02]. It has been shown that for some implementations it is possible to discover addresses of relevant objects by brute force attacks [SPP+04]. Information leakage attacks [SPP+04] are also a concern: the security provided by randomization relies on keeping the locations of objects secret; if locations are leaked out of the target process, the target can be compromised.

Another form of randomization is instruction set randomization [KKP03; BAP$^+$03]. The idea is to create process-specific randomized instruction sets by using a simple encoding of instructions, e.g. by XORing them with a random key, and decoding the instructions before executing them. Since the encoding key is secret, any code supplied by an attacker is decoded into a meaningless instruction sequence, when executed. This approach has a significant performance penalty, if implemented in software. Furthermore, it only blocks attacks that inject code into targets; attacks that merely change the control-flow or corrupt data are not detected. Attacks against instruction set randomization have been described in [SEP05].

Finally, it is important to note that the diversity of detection mechanisms that have been proposed makes it difficult for an attack to elude all of them. All of these mechanisms could be used as detectors in the Vigilante architecture. By generating SCAs, any detector can communicate useful information about the attack to all other computers in the system.

7.2.3 Survivability

Several systems have proposed mechanisms that, like Vigilante filters, allow vulnerable services to continue execution while being attacked.

Rinard et al. [RCD$^+$04] proposed *failure oblivious computing*. They use a C compiler that inserts runtime checks for illegal memory accesses using the C Range Error Detector [RL04]. Their system discards invalid memory writes, and redirects invalid memory reads to a pre-allocated buffer of values; they use heuristics to decide which values to use. While they show that several applications continue to execute normally when memory errors are masked in this way, it is not clear how this mechanism affects the correct execution of general programs.

Several systems proposed techniques that checkpoint/rollback executions to a previous execution point, upon detecting an attack. DIRA [SC05] is a compiler extension that can log updates to memory, and allows rolling back vulnerable services to the entry point of a function. Sidiroglou et al. [SLBK05] proposed using an emulator to execute code in regions where faults have been observed. When faults occur, their system rolls back memory writes and returns an error from the current function. Rx [QTSZ05] checkpoints processes periodically, and rolls them back to the latest checkpoint, when an error is detected. Rx then dynamically changes the execution environment based on the observed error. For instance, if a buffer overflow was observed, subsequent executions may allocate larger buffers to avoid the overflow. One limitation of the checkpoint/rollback approach is that rolling back past the point where I/O operations committed is problematic; for instance, state in disks or in processes that received network messages from the faulty process may become inconsistent. Performance is also a concern for two reasons. First, worm attack packets may be frequent, causing many rollbacks (Rx mitigates this concern by enforcing the changes to execution environment for a threshold interval, but it still discards them after that interval to reduce space and time overheads). Second, these systems require a detection mechanism to decide when to initiate a rollback, and high-coverage detection mechanisms are often expensive. Vigilante filters are more efficient than these techniques and they are less likely to affect the correct execution of the protected services.

Sidiroglou et al. [SLBK05] proposed generating patches automatically using a set of heuristics to modify vulnerable source code, for example, modifying the code to move vulnerable buffers to the heap. Their system still requires applications to stop for applying the patch, but after that they can continue executing. While they show that this approach works in some cases, it is difficult to provide guarantees on the semantics of the modified program.

Recently, several systems proposed using filters generated with host-based information. Buttercup [PCL$^+$04] proposed identifying the return address range used in worm attack messages and filtering messages that include such addresses. To reduce false positives, their system searches for the return address value starting at a predetermined offset in messages, and stops after a configurable number of bytes have been checked. While Buttercup requires these addresses to be externally specified, CTCP [HC04] and TaintCheck [NS05] proposed to obtain them automatically, by using the exact return address observed in attack messages. These systems can have false positives, because the 4 byte sequences used as a return address can appear in normal messages. The system can also have false negatives, since attackers can use a wide range of values of return addresses, by searching the address space of vulnerable applications for sequences of bytes that correspond to instructions that transfer control to the worm code [CSWC05].

ARBOR [LS05a] generates signatures based on the size of network messages and the fraction of non-ASCII characters in them. Its signatures also include host context: messages are dropped at specific code locations, and when specific call sequences are observed. ARBOR can still have false positives and false negatives. COVERS [LS05b] also generates signatures based on length of inputs and fraction of non-ASCII characters in them, but includes an input correlation mechanism to identify attack packets and the specific bytes in those packets that were involved in an observed security fault. Vigilante's SCA generation algorithm, performs this correlation in a more efficient way. COVERS uses information about the network protocol used by an application, to generate filtering conditions on specific fields of the protocol. Vigilante does not require network protocol information. COVERS does not provide guarantees on the rate of false positives or false negatives.

Several systems provide interesting alternatives to deploy Vigilante filters. IntroVirt [JKDC05] uses vulnerability-specific predicates to analyze the execution state of applications and operating systems running inside virtual machines. Like Vigilante filters, IntroVirt predicates can compute generic conditions, but they are generated manually for known vulnerabilities. By using virtual machine rollback and replay, IntroVirt is able to detect if vulnerabilities were exploited in the past. We could deploy Vigilante filters as IntroVirt predicates. Shield [WGSZ04] uses host-based filters to block vulnerabilities but these filters are generated manually. We could use Shield's infrastructure to deploy our filters.

7.3 Artificial immune systems

Several projects have contributed to the design of artificial immune systems. Cohen [Coh87] studied computer viruses, and Kephart et al. [KSSW97] designed a computer immune system targeted at viruses. Unlike viruses, worms spread automatically by exploiting software vulnerabilities. This led to a vulnerability centric-design in Vigilante that solves many of the problems faced by [KSSW97]. Hofmeyr [HF00] describes an artificial immune system inspired by natural immune systems. Their system can be applied to several domains, but it is not particularly well adapted to the problem of containing worm epidemics. One attack resilience principle inspired by natural systems is diversity [FSA97]. Interestingly, the argument that monocultures contribute to improved security has also been made [LSK06].

Several authors have proposed theoretical models for predicting characteristics of worm epidemics and for analyzing immunization systems [KW91; WKE00; SPW02; MSVS03; ZGGT03; CGK03; SMPW04; Sta04; VG05; GSSS05; GGK^{+}06]. Vigilante can be seen as a detailed design for an automatic artificial immune system that provides protection from worm attacks: we described how unknown worm attacks can be detected with broad coverage, how machines can safely share information about the attacks in a timely manner, and how machines can protect themselves efficiently.

Chapter 8

Conclusions

8.1 Summary

Systems to contain Internet worm epidemics must be deployed, because our society is increasingly dependent on computers connected to the Internet. Worm containment systems must be automatic, since worms infect computers much faster than humans can respond. However, automatic systems will not be widely deployed unless they are accurate. They cannot cause network outages by blocking innocuous traffic and they should be hard to evade.

Vigilante introduces an end-to-end architecture to automate worm containment. End hosts can contain worms accurately because they can perform a detailed analysis of attempts to infect the software they run. Vigilante introduces dynamic data-flow analysis: an algorithm that detects infection attempts with broad coverage. The algorithm detects the three most common infection techniques used by worms: code injection, edge injection and data injection, without requiring access to source code.

Vigilante introduces the concept of a self-certifying alert that enables a large-scale cooperative architecture to detect worms and to propagate alerts. Self-certifying alerts remove the need to trust detectors; they provide a common language to describe vulnerabilities and a common mechanism to verify alerts. Verifying SCAs is an effective way to discard any false positives generated by detectors. After detection, Vigilante uses an overlay to distribute SCAs in a resilient and timely manner.

Vigilante also introduces a new mechanism to generate host-based filters automatically by performing dynamic data and control flow analysis of attempts to infect programs. These filters can block mutations of the attacks observed by detectors and they produce a negligible performance degradation when deployed.

Our results show that Vigilante can contain real worms like Slammer, Blaster, CodeRed, and polymorphic variants of these worms, even when only a small fraction of the vulnerable hosts can detect the attack. Furthermore, Vigilante does not require any changes to hardware, compilers, operating systems or the source code of vulnerable programs. Therefore, Vigilante can be used to protect software as it exists today in binary form.

8.2 Future Work

There are several promising directions for future work. Recently, we observed that almost all worm attacks subvert the intended data-flow in a program [CCH06]. Based on this observation, we proposed a technique that can prevent both control and non-control data attacks by enforcing a simple safety property that we call *data-flow integrity*. This technique computes a data-flow graph for a vulnerable program using static analysis, and instruments the program to ensure that the flow of data at runtime is allowed by the data flow graph. We plan to integrate this new detector in the Vigilante architecture.

We are also working on combining static analysis techniques with our dynamic analysis to generate filters that can block more attack mutations. Analyzing more execution paths, besides the path identified by an SCA, and using static techniques such as program chopping [RR95] will yield more general filters.

The operational mechanism that we have used to verify SCAs could be augmented with a static version of verification. This can be seen as an application of *proof-carrying code* [NL96], where logic proofs of vulnerability are exchanged by machines.

Finally it interesting to consider integrating Vigilante with network telescopes [MVS01; MSVS04] and honeyfarms [VMC$^+$05]. By re-directing suspicious traffic to host-based detectors, network telescopes can help detect a worm outbreak sooner, yielding even better containment results than we have presented here.

Appendix A

Intel IA-32 Assembly Language

Table A.1 describes the small subset of Intel IA-32 [Int99] instructions used in the examples. Intel IA-32 CPUs store the address of the next instruction to be executed (i.e. the program counter) in the *eip* register. The *esp* register points to the top of the stack, which grows downwards in memory (i.e. towards lower memory addresses). *eax*, *ebx*, *ecx*, *ebp*, *esi* and *edi* are general purpose 32-bit registers. *al* and *dl* are 8-bit registers, corresponding to the least significant bytes of *eax* and *edx*, respectively. Arithmetic and logic instructions such as addition (*add*), subtraction (*sub*) and comparison (*cmp*) set the CPU flags (*eflags* register) according to the result. Conditional control-flow instructions take control-flow decisions based on the current values of the CPU flags.

Instruction	Description
mov dest,src	Move *src* to *dest*
mov dest,byte ptr [src]	Load byte from *src* memory address into *dest*
mov byte ptr [dest],src	Store *src* byte into memory address *dest*
sub dest,src	Subtract *src* from *dest* and store the result in *dest*
add dest,src	Add *src* to *dest* and store the result in *dest*
xor dest,src	Store in *dest* the bitwise exclusive OR between *dest* and *src*
lea dest,[src]	Load address specified by *src* into *dest*
cmp reg,val	Compare *reg* to *val*
ja address	Jump if above
je address	Jump if equal
jne address	Jump if not equal
push reg	Push *reg* onto the stack
pop reg	Pop the value on top of the stack into *reg*
call address	call function (pushes *eip* onto the stack)
ret	return from function (pops *eip* from the stack)

Table A.1: List of IA-32 assembly language instructions used in the examples.

References

[ABEL05] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow Integrity: Principles, implementations, and applications. In *ACM CCS*, Nov. 2005. 92

[ADLL05] Dzintars Avots, Michael Dalton, V. Benjamin Livshits, and Monica S. Lam. Improving software security with a C pointer analysis. In *ICSE*, May 2005. 90

[Aka00] Akamai. Press release: Akamai helps mcafee.com support flash crowds from iloveyou virus, May 2000. 47

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, techniques, and tools. *Prentice Hall*, 1986. 8

[Ban05] P. Bania. TAPiON. http://pb.specialised.info/all/tapion/. 2005. 17

[BAP$^+$03] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zov. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM CCS*, Oct. 2003. 96

[BCdJ$^+$06] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinic, Darek Mihocka, and Joe Chau. Framework for instruction-level tracing and analysis of program executuions. In *VEE*, June 2006. 28, 47, 60

[BCJ+05] Michael Bailey, Evan Cooke, Farnam Jahanian, David Watson, and Jose Nazario. The blaster worm: Then and now. *IEEE Security and Privacy*, 3(4), 2005. 7

[BDA00] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *ACM FDD0*, Dec. 2000. 67, 92

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, Oct. 2003. 95

[BDS03] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Usenix Security Symposium*, Aug. 2003. 95

[BEL75] R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT - a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, Apr. 1975. 90

[BFV05] John Bethencourt, Jason Franklin, and Mary Vernon. Mapping Internet sensors with probe response attacks. In *Usenix Security Symposium*, 2005. 19

[Bib77] K. J. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, MITRE, April 1977. 94

[BK00] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 10(46), May 2000. 92

[ble02] blexim. Basic integer overflows. *Phrack*, (60), Dec. 2002. 8

[Boc06] Bochs. Bochs ia-32 emulator. http://bochs.sourceforge.net. 2006. 28

[BPS00] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. In *Software Practice and Experience*, 2000. 90

[BSD05] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *Usenix Security Symposium*, Aug. 2005. 95

[BST00] A. Baratloo, N. Singh, and T. Tsai. Transparent runtime defense against stack smashing attacks. In *Usenix Technical Conference*, June 2000. 91

[Car04] Luca Cardelli. Type systems. *The Computer Science and Engineering Handbook. CRC Press.*, 2004. 90

[CBB$^+$01] Crispin Cowan, Matt Barringer, Steve Beattie, Gregh Kroah-Hartman, Mike Frantzen, and Jamie Lokier. Formatguard: Automatic protection from printf format string vulnerabilities. In *USENIX Security Symposium*, August 2001. 91

[CBJW03] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, August 2003. 91

[CBR03] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. Firewalls and Internet Security: Repelling the Wily Hacker. *Addison-Wesley*, 2003. 84

[CC04] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO-37*, Dec. 2004. 93, 94, 95

[CCC$^+$05] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-End Containment of Internet Worms. In *SOSP*, Oct. 2005. 93, 95

[CCCR04] Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Can we contain Internet worms? In *HotNets*, Nov. 2004. 93, 95

[CCH06] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data–flow integrity. In *OSDI*, Nov. 2006. 102

[CCR04] Miguel Castro, Manuel Costa, and Antony Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of The International Conference on Dependable Systems and Networks (DSN)*, June 2004. 48

[CDG$^+$02] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Security for structured peer-to-peer overlay networks. In *OSDI*, Dec. 2002. 44, 45, 48

[CER01] CERT. Cert advisory ca-2001-26 nimda worm. 2001. `http://www.cert.org/advisories/ca-2001-26.html`. 84

[CER05] CERT. Technical cyber security alerts. http://www.us-cert.gov. 2005. 1

[CGK03] Zesheng Chen, Lixin Gao, and Kevin Kwiat. Modelling the spread of active worms. In *IEEE INFOCOM*, Apr. 2003. 72, 99

[CH01] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *ICDCS*, Apr. 2001. 91, 92

[Coh87] Fred Cohen. Computer viruses, theory and experiments. In *Computers and Security*, volume 6, pages 22–35, 1987. 98

[CPG$^+$04] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding data lifetime via whole system simulation. In *Usenix Security Symposium*, Aug. 2004. 93

[CPM$^+$98] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Wadpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic detection and prevention of buffer-overrun attacks. In *USENIX Security Symposium*, Jan. 1998. 26, 91

[CSWC05] Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *ACM CCS*, Nov. 2005. 98

[CvdB05] Ramkumar Chinchani and Eric van den Berg. A fast static analysis approach to detect exploit code inside network flows. In *RAID*, Sept. 2005. 89

[CXN+05] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *DSN2005*, Jul. 2005. 95

[CXS+05] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Usenix Security Symposium*, Jul. 2005. 93

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, Aug. 1975. 50

[DKC+02] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *OSDI*, Dec. 2002. 38

[Dou02] John R. Douceur. The Sybil attack. In *IPTPS*, Mar. 2002. 44

[ds99] dark spyrit. Win32 buffer overflows. *Phrack*, 9(55), 1999. 9

[Dur02] Tyler Durden. Bypassing pax aslr protection. *Phrack*, (59), July 2002. 95

[DUYU03] Theo Detristan, Tyll Ulenspiegel, Yann_malcom, and Mynheer Superbus Von Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack*, 11(61), Aug. 2003. 17

[EAWJ02] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002. 38

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behaviour: A general approach to inferring errors in systems code. In *SOSP*, 2001. 91

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. In *IEEE Software*, Jan. 2002. 90

[ER89] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: an analysis of the Internet virus of november 1988. In *IEEE Symposium on Security and Privacy*, May 1989. 7

[FC03] Keir Fraser and Fay Chang. Operating System I/O Speculation: How two invocations are faster than one. In *USENIX Annual Technical Conference*, Jun. 2003. 70

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *IEEE Symposium on Security and Privacy*, 1996. 91

[FKF+03] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using system call information. In *IEEE Symposium on Security and Privacy*, 2003. 91

[For06] Forescout. Wormscout. http://www.forescout.com/wormscout.html. 2006. 84

[FSA97] Stephanie Forrest, Anil Somayaji, and David Ackley. Building diverse computer systems. In *HotOS*, 1997. 95, 99

[FSP+06] Prahlad Fogla, Monirul Sharif, Roberto Perdisci, Oleg Kolesnikov, and Wenke Lee. Polymorphic blending attacks. In *Usenix Security Symposium*, Aug. 2006. 88

[GEB02] G. Ganger, G. Economu, and S. Bielski. Self-securing network interfaces: What, why and how. Technical Report CS-02-144, Carnegie Mellon University, May 2002. 84

[GGK$^+$01] Fotis Georgatos, Florian Gruber, Daniel Karrenberg, Mark Santcroos, Henk Uijterwaal, and Rene Wilhelm. Providing Active Measurements as a Regular Service for ISPs. In *PAM*, Apr. 2001. `http://www.ripe.net/ttm`. 71

[GGK$^+$06] A. Ganesh, D. Gunawardena, P. Key, L. Massoulie, and J. Scott. Efficient quarantining of scanning worms: Optimal detection and coordination. In *IEEE INFOCOM*, Apr. 2006. 72, 85, 99

[GJM04] J. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *NDSS*, 2004. 91

[gr02] gera and riq. Advances in format string exploitation. *Phrack*, (59), July 2002. 8

[GSSS05] Jacob Goldenberg, Yuval Shavitt, Eran Shir, and Sorin Solomon. Distributive immunization of networks against viruses using the 'honey pot' architecture. In *Nature Physcis*, Dec. 2005. 99

[HB99] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. In *USENIX Windows NT Symposium*, July 1999. 36, 60, 78

[HC04] F. Hsu and Tzi-cker Chiueh. CTCP: A centralized TCP architecture for networking security. In *ACSAC*, 2004. 97

[HDK$^+$90] L. T. Heberlein, G. Dias, Levitt K, B. Mukerjeeand J. Wood, and D. Wolber. A network security monitor. In *IEEE Symposium on Research in Security and Privacy*, 1990. 16, 85

[Het00] H. W. Hethcote. The mathematics of infectious deseases. *SIAM Review*, 42(4):599–653, 2000. 14, 71

[HF00] Steven A. Hofmeyr and S. Forrest. Architecture for an artificial immune system. In *Evolutionary Computation*, 2000. 98

[HFC$^+$06] Alex Ho, Michael Fetterman, Christopher Clark, Andrew Warfield, and Steven Hand. Practical taint-based protection using demand emulation. In *EuroSys*, Apr. 2006. 95

[HOB99] Wei Hua, Jim Ohlund, and Barry Butterklee. Unraveling the mysteries of writing a winsock 2 layered service provider. *Microsoft Systems Journal*, May 1999. 30, 47

[HR05] Thorsten Holz and Frederic Raynal. Detecting honeypots and other suspicious environments. In *Workshop on Information Assurance and Security*, Jun. 2005. 19

[Int99] Intel. Intel architecture software developer's manual, volume 2: Instruction set reference. 1999. 29, 105

[JK97] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the International Workshop on Automatic Debugging*, May 1997. 92

[JKDC05] Ashlesha Joshi, Sam King, George Dunlap, and Peter Chen. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP*, Oct. 2005. 98

[JMG$^+$02] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002. 90

[Joh84] S. C. Johnson. Lint, a C program checker. In *Unix Programmer's Manual, 4.2. Berkeley Software Distribution Supplementary Documents*, 1984. 90

[jp03] jp. Advanced doug lea's malloc exploits. *Phrack*, (61), Sep. 2003. 8

[JPBB04] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast portscan detection using sequential hypothesis testing. In *IEEE Symposium on Security and Privacy*, May 2004. 85

[Jun06] J. Jung. Real-time detection of malicious network activity using stochastic models. In *PhD Thesis, Massachusetts Institute of Technology*, June 2006. 85

[JW04] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *Usenix Security Symposium*, Aug. 2004. 90

[K201] K2. Admmutate. http://www.ktwo.ca/security.html. 2001. 17

[KA94] J. O. Kephart and W. C. Arnold. Automatic extraction of computer virus signatures. In *International Virus Bulletin Conference*, Sept. 1994. 86

[Kat06] Ken Kato. Vmware backdoor I/O port. http://chitchat.at .infoseek.co.jp/vmware/backdoor.html. 2006. 19

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, Aug. 2002. 92

[KC03] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *HotNets*, Nov. 2003. 2, 17, 86

[Kin76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976. 57

[KK04] H. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium*, Aug. 2004. 2, 17, 57, 87

[KKM+05a] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *RAID*, Sept. 2005. 89

[KKM$^+$05b] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertsonand, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *Usenix Security Symposium*, Aug. 2005. 91

[KKP03] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *ACM CCS*, Oct. 2003. 96

[KSSW97] Jeffrey O. Kephart, Gregory B. Sorkin, Morton Swimmer, and Steve R. White. Blueprint for a computer immune system. In *International Virus Bulletin Conference*, Oct. 1997. 98

[KW91] Jeffrey O. Kephart and Steve R. White. Directed-graph epidemiological models of computer viruses. In *IEEE Symposium on Security and Privacy*, May 1991. 99

[LE01] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Usenix Security Symposium*, 2001. 90

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications using static analysis. In *USENIX Security Symposium*, Aug. 2005. 90

[LS05a] Zhenkai Liang and R. Sekar. Automatic generation of buffer overflow signatures: An approach based on program behavior models. In *ACSAC*, Dec. 2005. 98

[LS05b] Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *ACM CCS*, Nov. 2005. 98

[LSK06] Michael Locasto, Stelios Sidiroglou, and Angelos Keromytis. Software self-healling using collaborative application communities. In *NDSS*, Feb. 2006. 99

[Mad06] Anil Madhavapeddy. Creating High-Performance Statically Type-Safe Network Applications. In *PhD Thesis, University of Cambridge*, April 2006. 90

[Mir06] Mirage. Mirage networks. 2006. `http://www.miragenetworks.com`. 84

[Moc87] P. Mockapetris. Domain names: Concepts and facilities. Technical Report RFC-1034, Internet Engineering Task Force, Nov. 1987. 84

[MPS$^+$03] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer worm. *IEEE Security and Privacy*, 1(4), Jul. 2003. 1, 7, 14, 17, 63, 72

[MSB02] David Moore, Colleen Shannon, and J. Brown. Code-Red: A case study on the spread and victims of an Internet worm. In *ACM Internet Measurement Workshop*, Nov. 2002. 7

[MSVS03] David Moore, Colleen Shannon, Geoffrey Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *IEEE INFOCOM*, Apr. 2003. 2, 14, 72, 86, 99

[MSVS04] David Moore, C. Shannon, Geoffrey M. Voelker, and Stefan Savage. Network telescopes: Technical report. Technical Report CS2004-0795, UCSD, July 2004. 102

[MVS01] David Moore, G. M. Voelker, and Stefan Savage. Inferring Internet denial of service activity. In *Usenix Security Symposium*, Aug. 2001. 102

[Ner01] Nergal01. The advanced return-into-lib(c) exploits: Pax case study. *Phrack*, (58), 2001. 12

[NKS05] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, May 2005. 57, 88

[NL96] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *OSDI*, Oct. 1996. 102

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *POPL*, Jan. 2002. 90

[NS03] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, July 2003. 94

[NS05] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In *NDSS*, Feb. 2005. 93, 94, 97

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996. 8

[Pax99] Vern Paxson. Bro. a system for detecting network intruders in real time. *Computer Networks*, 31(23-24):2435–2463, December 1999. 16, 85

[PAX01] Pax team. 2001. `http://pax.grsecurity.net/`. 38, 39, 95

[PCL$^+$04] A. Pasupulati, J. Coit, K. Levitt, S. F. Wu, S. H. Li, J. C. Kuo, and K. P. Fan. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *NOMS*, Apr. 2004. 97

[PDL$^+$06] Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symposium on Security and Privacy*, 2006. 16, 88

[Per06] Perl security manual page. 2006. `http://www.perldoc.com`. 93

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc, Jan. 1998. 78

[Pro04] Neils Provos. A virtual honeypot framework. In *Usenix Security Symposium*, Aug. 2004. 19

[PSB06] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks. In *EuroSys*, Apr. 2006. 95

[QEM06] QEMU. Qemu open source processor emulator. 2006. `http://fabrice.bellard.free.fr/qemu/`. 28, 95

[QTSZ05] Fenq Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies: a safe method to survive software failures. In *SOSP*, Oct. 2005. 97

[RCD$^+$04] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee Jr. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, Dec. 2004. 96

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware*, November 2001. 15

[RFH$^+$01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *ACM SIGCOMM'01*, August 2001. 15

[RHR06] Costin Raiciu, Mark Handley, and David S. Rosenblum. Exploit hijacking: Side effects of smart defenses. In *SIGCOMM Workshops*, Sept. 2006. 76

[rix01] rix@hert.org. Writing ia32 alphanumeric shellcodes. *Phrack*, 11(57), Aug. 2001. 89

[RL04] O. Ruwase and M. Lam. A practical dynamic buffer overflow detector. In *NDSS*, Feb. 2004. 92, 96

[Roe99] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Conference on Systems Administration*, Nov. 1999. 16, 85

[RR95] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. In *FSE*, 1995. 102

[SBDB01] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *IEEE Symposium on Security and Privacy*, 2001. 91

[SC05] Alexey Smirnov and Tzi-cker Chiueh. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *NDSS*, Feb. 2005. 92, 96

[SCCD$^+$96] S. Staniford-Chen, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, and D. Zerkle. GrIDS: A graph-based intrusion detection system for large networks. In *Proceedings of the 19th National Information Systems Security Conference*, 1996. 84

[Sec02] SecurityFocus. Microsoft jvm class loader buffer overrun vulnerability. 2002. `http://www.securityfocus.com/bid/6134`. 90

[SEP05] Nora Sovarel, David Evans, and Nathanael Paul. Where's the FEEB? The effectiveness of instruction set randomization. In *Usenix Security Symposium*, Aug. 2005. 96

[SEVS04] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *OSDI*, Dec. 2004. 2, 17, 57, 87

[SF01] Peter Szor and Peter Ferrie. Hunting for metamorphic. In *International Virus Bulletin Conference*, September 2001. 17

[SH82] J. F. Shoch and J. A. Hupp. The worm programs: Early experience with a distributed computation. *Communications of the ACM*, 25(3):172–180, March 1982. 7

[SHM02] S. Staniford, J. Hoagland, and J. McAlerney. Practical automated detection of stealthy portscans. *Journal of Computer Security*, 10:105–136, 2002. 85

[SII05] Yoichi Shinoda, Ko Ikai, and Motomu Itoh. Vulnerabilities of passive internet threat monitors. In *USENIX Security Symposium*, Aug. 2005. 19

[SJB04] Stuart Schechter, Jaeyeon Jung, and Arthur Berger. Fast detection of scanning worm infections. In *RAID*, Sep. 2004. 85

[SLBK05] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *Usenix Annual Technical Conference*, Apr. 2005. 96, 97

[SLD04] G. Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS XI*, Oct. 2004. 93, 95

[SM04] Colleen Shannon and David Moore. The spread of the Witty worm. *IEEE Security and Privacy*, 2(4), Jul. 2004. 7, 15

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *ACM SIGCOMM'01*, August 2001. 15

[SMPW04] Stuart Staniford, David Moore, Vern Paxson, and Nicholas Weaver. The top speed of flash worms. In *WORM*, Oct. 2004. 99

[Spa89] Eugene H. Spafford. The Internet worm: Crisis and aftermath. *Communications of the ACM*, 32(6):678–687, June 1989. 7

[SPE] SPEC. Specweb99 benchmark. `http://www.spec.org/osg/web99`. 67, 79

[SPP$^+$04] Hovav Shacham, Mattew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address space randomization. In *ACM CCS*, Oct. 2004. 95

[SPW02] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the Internet in your spare time. In *USENIX Security Symposium*, Aug. 2002. 2, 14, 15, 99

[Sta04] Stuart Staniford. Containment of scanning worms in enterprise networks. *Journal of Computer Security*, 2004. 85, 99

[STFW01] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Usenix Security Symposium*, Aug. 2001. 90

[TC05] Y. Tang and S. Chen. Defending against Internet worms: A signature-based approach. In *IEEE INFOCOM*, March 2005. 88

[TK02a] Thomas Toth and Christopher Kruegel. Accurate buffer overflow detection via abstract payload execution. In *RAID*, Oct. 2002. 89

[TK02b] Thomas Toth and Christopher Kruegel. Connection-history based anomaly detection. In *IEEE Information Assurance Workshop*, June 2002. 85

[TPC99] TPC. TPC-C online transaction processing benchmark. 1999. `http://www.tpc.org/tpcc/default.asp`. 67, 78

[Ven01] Vendicator. Stack shield technical info. http://www.angelfire.com /sk/stackshield. 2001. 91

[VG05] Milan Vojnović and Ayalvadi Ganesh. On the race of worms, alerts and patches. In *WORM*, Nov. 2005. 72, 99

[VMC$^+$05] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity and containment in the potemkin virtual honeyfarm. In *SOSP*, Oct. 2005. 70, 102

[WCS05] Ke Wang, Gabriela Cretu, and Salvatore J. Stolfo. Anomalous payload-based worm detection and signature generation. In *RAID*, Sept. 2005. 88

[Wei84] M. Weiser. Program slicing. *IEEE Transaction on Software Engineering*, 10(4):352–357, 1984. 57

[WESP04] N. Weaver, D. Ellis, S. Staniford, and V. Paxson. Worms vs. Perimeters: The Case for Hard-LANs. In *Hot Interconnects 2*, Aug. 2004. 85

[WFBA00] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *NDSS*, Feb. 2000. 90

[WGSZ04] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *ACM SIGCOMM*, Aug. 2004. 98

[Wil02] Matthew M. Williamnson. Throttling viruses: Restricting propagation to defeat mobile malicious code. In *ACSAC*, Dec. 2002. 16, 86

[Win93] Glyn Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993. 50

[WK03] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS*, Feb. 2003. 26, 65, 92

[WKE00] C. Wang, J. Knight, and M. Elder. On computer viral infection and the effect of immunization. In *ACSAC*, Dec. 2000. 99

[WKO05] David Whyte, Evangelos Kranakis, and P. C. Van Oorschot. Dns-based detection of scanning worms in an enterprise network. In *NDSS*, Feb. 2005. 84

[WPLZ06] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. Sigfree: A signature-free buffer overflow attack blocker. In *Usenix Security Symposium*, Aug. 2006. 89

[WPSC03] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A taxonomy of computer worms. In *WORM*, Oct. 2003. 14

[WS02] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM CCS*, 2002. 91

[WSP04] N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *USENIX Security Symposium*, Aug. 2004. 2, 16, 85

[XA05] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *POPL*, Jan. 2005. 90

[XKI03] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *SRDS*, Oct. 2003. 95

[YGBJ05] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha. An architecture for generating semantics aware signatures. In *Usenix Security Symposium*, Aug. 2005. 88

[YTEM04] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *OSDI*, Dec. 2004. 90

[Z0m00] Z0mbie. Real Permutation Engine. http://vx.netlux.org/vx.php?id=er05. 2000. 17

[ZCB96] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *IEEE INFOCOM*, Mar. 1996. 70

[ZGGT03] Cliff Changchun Zou, Lixin Gao, Weibo Gong, and Don Towsley. Monitoring and early warning for Internet worms. In *ACM CCS*, Oct. 2003. 72, 99