Vigilante: End-to-End Containment of Internet Worm Epidemics

MANUEL COSTA
University of Cambridge and Microsoft Research
JON CROWCROFT
University of Cambridge
and
MIGUEL CASTRO, ANTONY ROWSTRON, LIDONG ZHOU, LINTAO ZHANG, and PAUL BARHAM
Microsoft Research

Worm containment must be automatic because worms can spread too fast for humans to respond. Recent work proposed network-level techniques to automate worm containment; these techniques have limitations because there is no information about the vulnerabilities exploited by worms 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 this 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 and all its polymorphic mutations that follow the execution path identified by the SCA.

Our results show that Vigilante can contain fast-spreading worms that exploit unknown vulnerabilities, and that Vigilante's filters introduce a negligible performance overhead. 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 current software binaries.

M. Costa is also affiliated with Microsoft Research.
Authors' addresses: M. Costa (corresponding author), J. Crowcroft, Computer Laboratory, University of Cambridge, The Old Schools, Trinity Lane, Cambridge CB2 1TN, UK; email: manuelc@microsoft.com; M. Castro, A. Rowstron, L. Zhou, L. Zhang, P. Barham, Microsoft Research, 7 J. J. Thomson Avenue, Cambridge, CB3 OFB, UK.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection

General Terms: Security, Reliability, Performance, Algorithms

Additional Key Words and Phrases: Worm containment, program analysis, dynamic data-flow analysis, self-certifying alerts, vulnerability condition slicing

ACM Reference Format:

Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., and Barham, P. 2008. Vigilante: End-to-End containment of Internet worm epidemics. ACM Trans. Comput. Syst. 26, 4, Article 9 (December 2008), 68 pages. DOI=10.1145/1455258.1455259 http://doi.acm.org/10.1145/1455258.1455259

1. INTRODUCTION

Worms[1]—programs that self-replicate automatically over computer networks—constitute a serious threat to hosts connected to the Internet. They exploit software defects to infect remote hosts without any user intervention, and they can use their victims for malicious activities such as corrupting data, sending unsolicited electronic mail messages, generating traffic for distributed denial-of-service attacks, and stealing information. The first experiments with programs similar to modern worms were reported in 1982 [Shoch and Hupp 1982]. 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 [Spafford 1989; Eichin and Rochlis 1989]; since then, several major outbreaks have caused serious financial losses and disruption of critical software services [Moore et al. 2003, 2002; Bailey et al. 2005; Shannon and Moore 2004].

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 [CERT 2005] suggests that we are still far from producing defect-free software: Figure 1 shows that exploitable software defects (also referred to as vulnerabilities) continue to be discovered at a rate of several thousand per year.

Another avenue to solve the worm problem is containment. Containment systems are built on a premise accepting that software has defects that can be exploited by worms, and they strive to contain a worm epidemic within a small fraction of the vulnerable hosts. Containment systems need to be fully automatic because worms can spread far faster than humans can respond [Moore et al. 2003; Staniford et al. 2002]. Recent work on automatic containment [Kim and Karp 2004; Singh et al. 2004; Kreibich and Crowcroft 2003; Weaver et al. 2004] 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

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

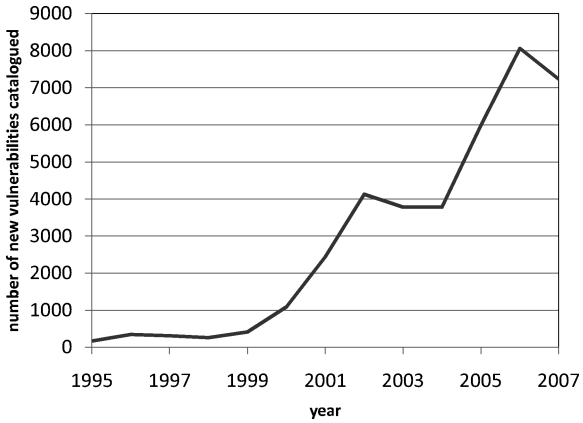

Fig. 1. Number of new vulnerabilities cataloged by CERT per year, from 1995 to 2007.

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 article presents Vigilante, a new system to contain worm epidemics automatically that addresses these limitations with an *end-to-end* architecture. An end-to-end architecture can use information about the vulnerable programs running at end-hosts. This information, which is not available to network-level approaches, can be leveraged to detect worms even when they exhibit normal network traffic patterns. This detection can verify unequivocally that a host is running vulnerable software and 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.

Figure 2 illustrates automatic worm containment with Vigilante. In Vigilante, hosts detect worms by instrumenting network-facing programs 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.

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 program and can be inexpensively verified. By verifying an SCA, a host can determine with certainty that a program is vulnerable; the verification procedure has no false positives. SCAs enable

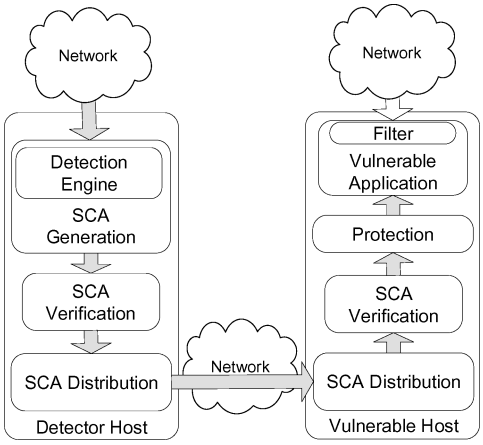

Fig. 2. Automatic worm containment in Vigilante.

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 and highly accurate detection engines, 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 [Provos 2004]; 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 provide a common format 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 program.

Alerted hosts protect themselves by generating filters that block worm messages before they are delivered to a vulnerable program. 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. Since each vulnerable host runs this procedure 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 have no false positives, and block all polymorphic worm mutations [Szor and Ferrie 2001] that follow the same execution path to gain control.

To validate the system, we implemented the algorithms on Intel IA-32 hosts running the Windows operating system. We tested Vigilante with a broad range of synthetic attacks and with three infamous real worms: Slammer, CodeRed, and Blaster. We also simulated the overall behaviour of the system when deployed on the Internet using parameters extracted from our implementation, and from Internet measurements of the Slammer, CodeRed, and Blaster 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 programs 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 current software binaries. Vigilante makes the following contributions:

—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.

The rest of this article is organized as follows. Section 2 describes the design and implementation of the dynamic data-flow analysis detection algorithm, and discusses the importance of using a diverse set of detection mechanisms in Vigilante. Section 3 introduces the concept of SCA and describes procedures to verify, generate, and distribute SCAs. Section 4 presents the design and implementation automatic filter generation with vulnerability condition slicing. Section 5 presents our experimental results. Section 6 describes related work. We conclude in Section 7 and discuss some directions for future work.

2. DETECTION

The first step to contain the outbreak of an unknown worm is to detect it. Vigilante detects worms by analyzing the execution of vulnerable programs at

end-hosts; it detects a worm when it is about to infect a running program. Using host-based detectors enables Vigilante to detect worms that have normal network traffic patterns, since at some point they still need to infect their victims. To be effective, worm detectors need to have high coverage and generate few false positives. This section introduces dynamic data-flow analysis, a new host-based detection algorithm that achieves these goals. Before describing the algorithm, we analyze how worms infect remote hosts, since the infection techniques used by worms guided our design of dynamic data-flow analysis.

2.1 Infection

Remotely infecting a host requires coercing the host 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 [One 1996], arithmetic overflows [blexim 2002], memory management errors [jp 2003], and incorrect handling of format strings [gera and riq 2002] are among the most common types of vulnerabilities. Worms exploit these vulnerabilities to gain control of remote programs 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 these three kinds of infection.

The next sections discuss these three infection mechanisms in detail. To facilitate the discussion, we use a program with a simple stack-based buffer overflow vulnerability [One 1996], shown in Figure 3, but it is important to note that all other types of vulnerabilities enable the same types of successful infection. The program in Figure 3 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 *user_id*, respectively. The code assumes fields in the message are separated by the newline character. The program is vulnerable because it does not check if the first field of the message fits in the stack-based *request* buffer. Consequently, the worm can send a message with a large first field and overwrite the stack frame. This defect can be exploited to infect the program with code injection, edge injection, and data injection.

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 3, 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

```
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;
}
```

Fig. 3. Vulnerable code in the C++ programming language. The code has a buffer overflow vulnerability enabling code injection, edge injection, and data injection attacks.

would ensure that, upon returning from the function, the process would start to run worm code.

The details of the attack can be understood by analyzing the vulnerable program in IA-32 assembly language, as shown in Figure 4. 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.[2]

[2]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 [dark spyrit 1999].

```
 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 user_id 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 user_id 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 user_id 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
```

Fig. 4. Vulnerable program in IA-32 assembly language (compiled from the source code in Figure 3). The code is vulnerable to code injection, edge injection, and data injection attacks.

2.1.2 *Edge Injection.* Infecting a remote host 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 [nergal 2001]. To use this type of attack on the program in Figure 3, 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 this 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.

2.1.3 *Data Injection.* Finally, infecting a remote host 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 behavior of the program without injecting any code or forcing any control-flow transfers.

Using again the example in Figure 3, 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 (i.e., 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 *user_id* parameter. This parameter is in the function's stack frame and can be easily overwritten. Therefore, all the worm need do is to overwrite *user_id* 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.

2.2 Dynamic Data-Flow Analysis

To remotely exploit software defects with the infection techniques described in the previous section, a worm needs to send messages that trigger the execution of the defective code at the target hosts. 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 this data; therefore, it is a form of information-flow security [Fenton 1974a, 1974b,1973; Denning 1976]. Like the other mechanisms introduced by Vigilante, dynamic data-flow analysis is based on analyzing the vulnerable code infected by worms. This approach is more effective than analyzing the worm's code or the worm's behavior because the vulnerable code is not controlled by the worm.

```
OnReceivedNetworkMessage(address, size)
    for i ← 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
```

Fig. 5. Dynamic data-flow analysis algorithm.

2.2.1 *Algorithm.* The dynamic data-flow analysis algorithm, shown in pseudocode[3] in Figure 5, 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 as *dirty*. Then, the algorithm tracks all movements of this 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.

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 Section 2.1, worms can infect a process using three types of attack: 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

[3]We use the pseudocode notation from Cormen et al. [1990]; in particular, lines starting with a triangle are comments.

following situations:

(1) execution of dirty data;
(2) loading of dirty data into the program counter; and
(3) passing of dirty data in arguments of security-sensitive functions.

Preventing execution of dirty data is important because the data has come 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 of 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 aforesaid conditions, 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 a 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 4 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 Section 2.1).

When the code starts to execute, the *esp* register points to the return address saved by the *call* instruction 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 6 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

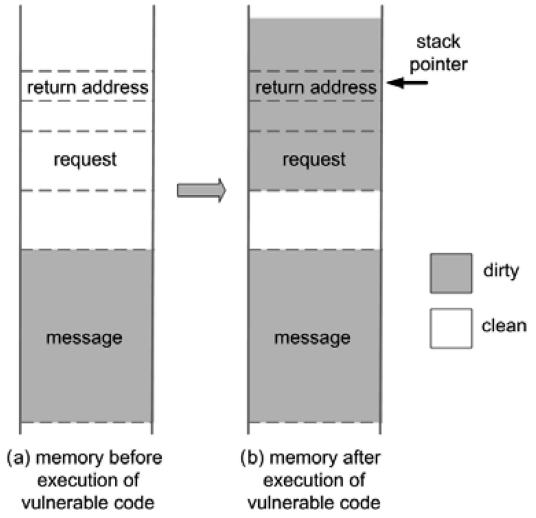

Fig. 6. Example of worm detection with dynamic data-flow analysis. The figure shows the memory when: (a) a message is received and the vulnerable code is about to execute; and (b) after the vulnerable code executes and overwrites the return address in the stack. Greyed areas indicate dirty memory regions.

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.

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 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 vulnerabilities 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 easy to bypass [Wilander and Kamkar 2003]. For instance, mechanisms that protect return addresses on the stack [Cowan et al. 1998] can be bypassed by overwriting function pointers.

Third, dynamic data-flow analysis works on unmodified program binaries. 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 arbitrary binaries. Furthermore, it works even in cases where program analysis is hard, such as programs with self-modifying or 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 [Holz and Raynal 2005; Bethencourt et al. 2005; Shinoda et al. 2005]. 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. The worm could then refrain from performing malicious actions inside the detection environment, thus avoiding detection.

Dynamic data-flow analysis has several limitations. First, it may generate false positives because programs may perform safety checks on data received from the network before using this 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 programs, it still generates a security trap on 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 Section 5 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 need 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 are not affected by this restriction. Even 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 Section 3 a verification mechanism to discard any false positives generated by detectors in the Vigilante containment system.

Dynamic data-flow analysis may also have false negatives, that is, there are several attacks that it cannot detect. It 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 simply due to unintentional development mistakes. Dynamic data-flow analysis also cannot detect software configuration errors such as weak passwords. Accessing a host 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.

2.2.2 *Implementation.* The dynamic data-flow analysis algorithm 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 also be implemented by changing compilers to emit instructions that inline the algorithm with the program's instructions. Finally, it can also be implemented by using processor emulators [Bochs 2006; QEMU 2006] to analyze each instruction as it is emulated.

While all the aforementioned implementations are viable, we chose to implement the algorithm with a dynamic binary rewriting tool, because this allows us to run the algorithm on unmodified binaries with reasonable performance. Specifically, we have used the Nirvana runtime instrumentation engine [Bhansali et al. 2006] 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.

The dynamic data-flow analysis implementation instruments every data movement instruction for Intel IA-32 [Intel 1999] 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 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) [Hua et al. 1999]. 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 7 illustrates the components inside a process running under the control of the dynamic data-flow analysis detector. The code and data layout inside

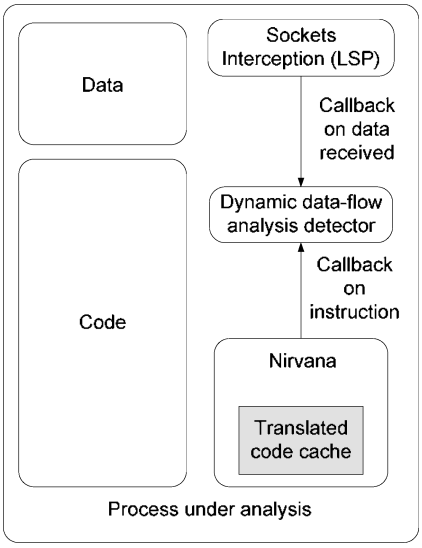

Fig. 7. Components inside a process running under the control of the dynamic data-flow analysis detector.

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 noninstrumented execution).

2.3 Diversity of Detection Mechanisms

Vigilante can use other host-based detectors, besides dynamic data-flow analysis. We believe it is important 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 use fingerprinting techniques [Holz and Raynal 2005; Bethencourt et al. 2005; Shinoda et al. 2005] that evade the detectors.

3. SELF-CERTIFYING ALERTS

Detecting a worm outbreak is not sufficient to contain it: Vulnerable hosts that have not yet been infected need to be protected. The first step to achieve this is to inform the vulnerable hosts about the worm outbreak. To do this, Vigilante

detectors generate self-certifying alerts (SCAs). An SCA is a new type of security alert that claims that a program is vulnerable to infection, and includes enough information for hosts to verify this claim. Since SCAs can be verified, any host in Vigilante can independently act as a detector. This allows hosts to share the detection load (e.g., different organizations may run different detectors), and makes it harder for a worm to evade detection because it is difficult to know where detectors are deployed. Using SCAs, hosts cooperate to contain an outbreak without having to trust each other. In this section we describe the format of SCAs, as well as the mechanisms to verify, generate, and distribute this new type of security alert.

3.1 Alert Types

An SCA proves that a program is vulnerable by describing how to exploit the program 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 program, cause it to reach a disallowed state. SCAs are verified by sending the messages to the program 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, described next.

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

—*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 program.

—*Arbitrary function argument* alerts identify data-injection vulnerabilities that allow worms to change the value of arguments to critical functions (e.g., 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 program.

These alert types are general. They demonstrate how the worm can gain control by using the external messaging interface to a program, 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; hence different types of detection engines can cooperate to contain an outbreak. This cooperation reduces the overall false negative rate of the system.

The three types of SCAs have a common format: an identification of the vulnerable program, 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 alert.

Program: 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,DC,C9,B0,42,EB,
0E,01,01,01,01,01,01,01,70,AE,42,01,70,AE,42,...

Fig. 8. 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.

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 this function, and where to put the corresponding actual argument value in the sequence of messages.

Figure 8 shows an example arbitrary execution control SCA generated for the Slammer worm. The SCA identifies the vulnerable program as Microsoft SQL Server version 8.00.194 and the alert type as an arbitrary execution control. The 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.

3.2 Alert Verification

Verifying an SCA entails reproducing the infection process by sending the sequence of messages in the alert to a vulnerable program. 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 program 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 programs. Each program 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 tool [Hunt and Brubacher 1999]. 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 programs, we can instrument any

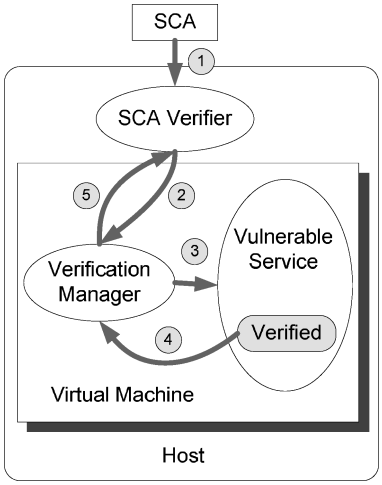

Fig. 9. SCA verification.

program. 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 9 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 program. Then it modifies the sequence of messages in the SCA to trigger execution of `Verified` when the messages are sent to the vulnerable program. The modifications involve changing the byte string at the offset of the message specified in the verification information according to 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 program. 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 5 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 program 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 program is vulnerable to the exploit described in the SCA. A successful verification shows that attackers can control a vulnerable program 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 program 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 program 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 program. 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 [Elnozahy et al. 2002; Dunlap et al. 2002] that we could leverage.

3.3 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 programs, but some details are necessarily detection-engine-specific.

To generate SCAs, hosts log messages and the networking endpoints where they are received during program 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 nonexecutable (*NX*) pages [PAX 2001] algorithm, which we describe next, and the dynamic data-flow analysis detection algorithm described in Section 2. 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 requires access to source code.

3.3.1 *Using Nonexecutable Pages.* The first detection engine uses nonexecute 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 [PAX 2001]. This detector can be used to generate arbitrary execution control or arbitrary code execution SCAs, described 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 multimessage exploits. In such a 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 has no further messages.

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 logged messages in the first candidate SCA when the log size is smaller than a threshold (e.g., 5).

3.3.2 *Using Dynamic Data-Flow Analysis.* Dynamic data-flow analysis can be used to generate the three types of alerts discussed in Section 3.1. 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 2.2.1: 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, explained as follows.

—If dirty data is about to be loaded into the program counter, it generates an arbitrary execution control SCA.
—If dirty data is about to be executed, it generates an arbitrary code execution SCA.
—If a critical argument to a critical function is dirty, it generates an arbitrary function argument SCA.

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 multimessage exploits, the detector follows the same search strategy to compute candidate SCAs as the detector based on nonexecutable pages.

We will use the vulnerable code in Figure 10 to illustrate SCA generation using dynamic data-flow analysis (the source code for the program is shown in Figure 3) 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

```
 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
```

Fig. 10. Example of SCA generation with vulnerable program in IA-32 assembly language (compiled from the source code in Figure 3).

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. Instruction 22 loads the next byte of the message into *dl*, which becomes tagged with 102. The byte is then compared

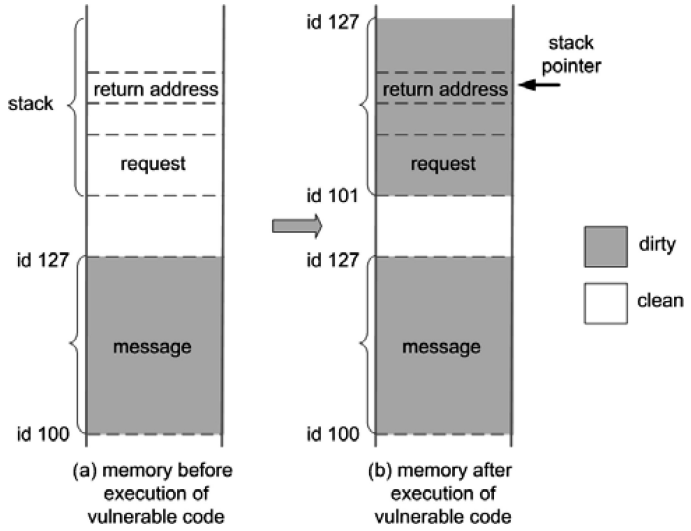

Fig. 11. Example of SCA generation with dynamic data-flow analysis. The figure shows the memory when: (a) a message is received and the vulnerable code is about to execute; and (b) after the vulnerable code executes and overwrites the return address in the stack. Greyed areas indicate dirty memory regions and the identifiers of dirty data are shown on the left.

with the newline character (*0x0A*), and the loop continues if the newline was not reached. The loop eventually overwrites the stored return address.

Figure 11 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 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 Section 2, 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 rewriting. SCAs address both of these issues: Verification eliminates false positives and the cooperative detection architecture spreads the detection load.

3.4 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 [Castro et al. 2002] to broadcast SCAs.

Vigilante uses flooding to broadcast SCAs to all hosts in the overlay: Each host sends the SCA to all its overlay neighbors. 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 $15 \times log_{16}N$ neighbors and the expected path length between two hosts is approximately $log_{16}N$. Since each host has a significant number of neighbors, 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 neighbors.

The secure overlay also includes defenses against active attacks. It prevents sybil attacks [Douceur 2002] by requiring each host to have a certificate signed by a trusted offline certification authority in order to participate in the overlay [Castro et al. 2002]. 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 neighbors [Castro et al. 2002]. These constraints completely specify the set of neighbors of any host for a given overlay membership. Each host establishes authenticated and encrypted connections with its neighbors using the certified public keys. Since compromised hosts cannot choose their *hostIds*, they are not free to choose their neighbors and 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 neighbor. The first technique prevents attacks that flood variants of old SCAs and the second prevents attacks that flood bogus SCAs to all hosts in the overlay. Since hosts only accept SCAs received over the authenticated connections to their neighbors, the third technique bounds the computational overhead that compromised hosts can impose on their neighbors. It is effective because the constraints on neighbor identifiers make it hard to change neighbors.

Requiring hosts to verify SCAs before forwarding raises some issues. Some hosts may be unable to verify valid SCAs because either 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 neighbors per host and to ensure 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 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 described before, we verify SCAs from different neighbors 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 neighbors 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 this information 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 neighbors.

Our preferred technique to hide information about the overlay topology from the worm is to run an overlay with superpeers. The superpeers 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 programs to verify SCAs efficiently. The superpeers form a secure Pastry overlay as we described. Each ordinary host connects to a small number q of superpeers (e.g., $q = 2$) that are completely specified by the host's identifier. This prevents leaking information about vulnerable hosts because all neighbors of compromised hosts are superpeers that do not run vulnerable software.

An overlay with superpeers is also more resilient to denial-of-service attacks. First, we can give priority to verification of SCAs sent by superpeers. Since superpeers are less likely to be compromised than ordinary hosts, this is an effective defence against denial-of-service attacks that bombard hosts with SCAs. Additionally, superpeers may be well-connected nodes with 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 superpeers 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 antivirus companies to distribute signatures [Akamai 2000]. However, it should be noted that alerts could be distributed over other broadcast/multicast channels (e.g., channels used to broadcast video).

3.5 Implementation

The implementation of SCA generation uses techniques similar to the ones described in Section 2, for the implementation of the dynamic data-flow

analysis detector. Vigilante intercepts socket operations, using a layered service provider [Hua et al. 1999], 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 [Bhansali et al. 2006] 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 Section 2 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 3.3.2).

SCAs are verified inside a Virtual PC 2004 virtual machine (VM) to isolate any side effects of the verification process (see Figure 9). 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 programs. The verification manager injects a dynamic link library (DLL) into each program by creating a new thread that loads the DLL. The DLL includes the `Verified` function, as well as 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 3.2, and waits on a synchronization object. If the SCA is valid, the `Verified` function is called and sets the synchronization object, signaling 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 Castro et al. [2004, 2002]. We used a small real network to evaluate the distribution of SCAs. To understand the behavior of Vigilante on the Internet, we simulated the distribution system using topologies from the secure version of the overlay [Castro et al. 2002]. The simulations also used measurements from real worm outbreaks and from our implementation of Vigilante.

4. PROTECTION

The last crucial step to contain a worm outbreak is to protect vulnerable hosts that have not yet been infected. 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 less expensive 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 behavior 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 polymorphic mutations [Szor and Ferrie 2001] of the worm attack. In this section 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 low overhead.

4.1 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 [1975]. 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 [Winskel 1993]. 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.

4.2 Vulnerability Condition Slicing

4.2.1 *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 is updated when an instruction uses a dirty processor flag, transfers control to an address read from a dirty location, or reads/writes from/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 12 shows the vulnerability condition slicing algorithm in pseudocode. 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 the 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 outcome of the test on the flag is recorded by negating the opcode if the jump is not *taken*; negating the opcode denotes replacing it with the opcode for the converse condition, for example, replacing the opcode for `je` (jump if equal) with the opcode for `jne` (jump if not equal). 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.

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

ONRECEIVEDNETWORKMESSAGE(address, size)
    for i ← 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(¬opcode, GETTAG(flag)))

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

Fig. 12. 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.

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`.

```
 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
```

Fig. 13. Example of filter generation with vulnerable program in IA-32 assembly language (compiled from the source code in Figure 3).

We will use the vulnerable code shown in Figure 13, as well as the corresponding arbitrary execution control SCA from Section 3.3, to illustrate the filter generation procedure (the mechanics of attacks on this code were described in Section 2.1). 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 14(a) shows the data-flow graph associated with the flags at this point. The filter condition

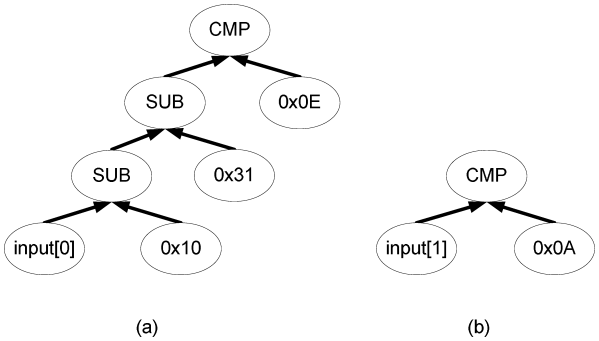

Fig. 14. Data-flow graphs for flags controlling conditional jumps: (a) when instruction `ja 45` is executed; and (b) when the instruction `jne 21` is executed. Both instructions are executed by the vulnerable program in Figure 13.

is updated to `input[0] - 0x10 - 0x31<=0x0E` after instruction 9, because the conditional jump *ja* is not taken.

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]`$\neq$`0x0A` for `i >= 1`, because *dl* is compared with `0x0A` and a conditional jump continues the loop if they are not equal; Figure 14(b) shows the data-flow graph associated with the flags when these control-flow decisions are taken.

Figure 15 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.

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

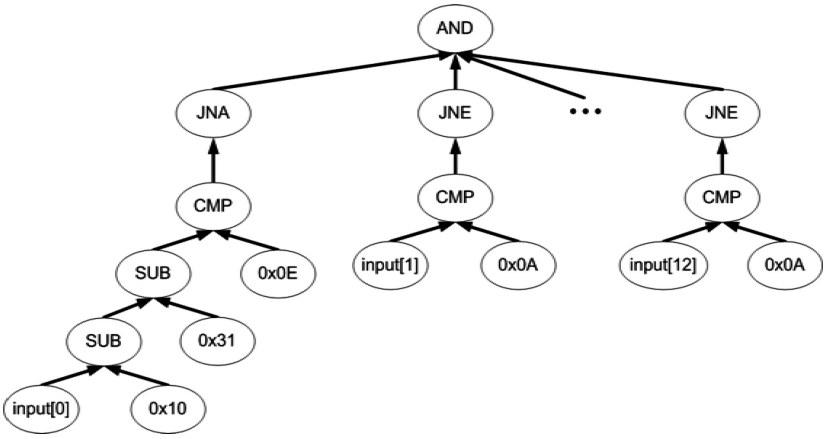

Fig. 15. Filter condition for an edge injection attack on the program in Figure 13. The filter blocks mutations of the attack; it matches any attack message with the first byte in the allowed range and the subsequent bytes different from 0x0A, up until the bytes that overwrite the return address on the stack.

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 16 shows the translation of the filter condition in Figure 15 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 the 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 totally different from filters that block messages that contain a particular string [Kim and Karp 2004; Singh et al. 2004] or sequence of strings [Newsome et al. 2005]. They can capture arbitrary computations on the values of the input messages.

This algorithm can be seen as a form of program slicing [Weiser 1984]. It identifies a subset of instructions in the program that compute those control-flow decisions that lead to successful attacks. The instructions captured in the data-flow graphs in Figure 14 are a subset of the instructions of the vulnerable program shown in Figure 4. 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 [King 1976]: Simultaneously with the concrete execution of the vulnerable program, the algorithm executes symbolically the instructions that process dirty data.

```
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
```

Fig. 16. Filter code generated automatically for the filter condition in Figure 15. The filter blocks mutations of an edge injection attack on the vulnerable program shown in Figure 13. The code to save registers and to set-up a separate stack is omitted for brevity.

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 earlier 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

require a longer than necessary sequence of nonzero 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 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.

4.2.2 *Implementation.* The implementation of filter generation uses techniques similar to the ones described in Section 2, 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.

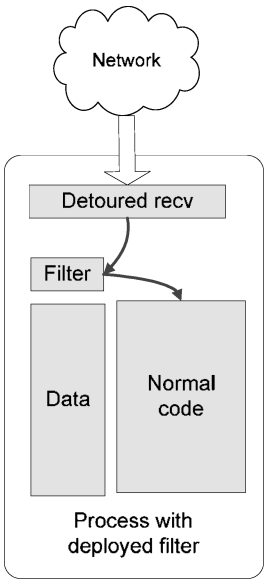

Fig. 17. Components inside a process with a filter deployed by Vigilante. Vigilante intercepts socket functions to process network messages with the filter code. When the filter matches a message, it is dropped; otherwise, it is handed over to the normal code.

The implementation intercepts socket operations, and tags each received byte with a unique data-flow graph that identifies the byte. We use Nirvana [Bhansali et al. 2006] 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 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 16. 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 [Hunt and Brubacher 1999] runtime instrumentation package. The interception mechanism used by Detours has 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 restarting the vulnerable program. 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 17 shows the components inside a vulnerable process, after a filter is deployed.

5. EVALUATION

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

5.1 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 programs and had a high impact on the Internet.

Slammer infected approximately 75,000 Microsoft SQL Servers. So far, it has been the fastest worm in history [Moore et al. 2003]. During its outbreak, the number of infected hosts 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 more slowly 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 hosts. 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.

5.2 Detection

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

The synthetic attacks were based on a testbed of 18 buffer overflow attacks described in Wilander and Kamkar [2003]. 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, detailed next.

Table I. Real Attacks Detected by Dynamic Data-Flow Analysis

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

Table II. Synthetic Attacks Detected by Dynamic Data-Flow Analysis

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

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 II shows the results for synthetic attacks. All 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% of the attacks [Wilander and Kamkar 2003]. Even if all the techniques used by the tools tested in Wilander and Kamkar [2003] 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

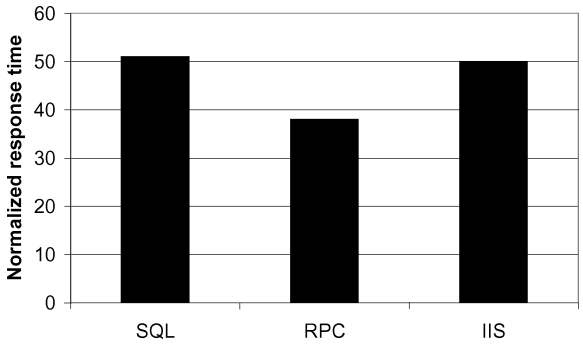

Fig. 18. Runtime overhead of running the dynamic data-flow analysis detector.

Windows RPC service. For each vulnerable program we measured the average response time of one hundred requests. For SQL Server the requests were generated with transactions from the TPC-C benchmark [TPC 1999]. 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 [SPEC 2008] 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 look up an RPC interface; these requests are also CPU bound. Figure 18 shows the overhead for each of the experiments for the three vulnerable programs. 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 deployments. The largest contributors to the overhead are the Nirvana rewriting mechanism and the disassembler used to decode instructions. Both of these mechanisms can be optimized; for instance, DynamoRIO [Bruening et al. 2001] provides much faster rewriting, and we also plan to optimize the detector by caching decoded instructions. However, as Section 5.3 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.

5.3 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 until 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 nonexecute 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.

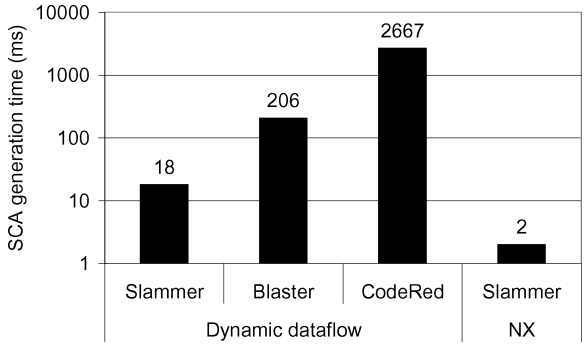

Fig. 19. SCA generation time in milliseconds for real worms using two detectors.

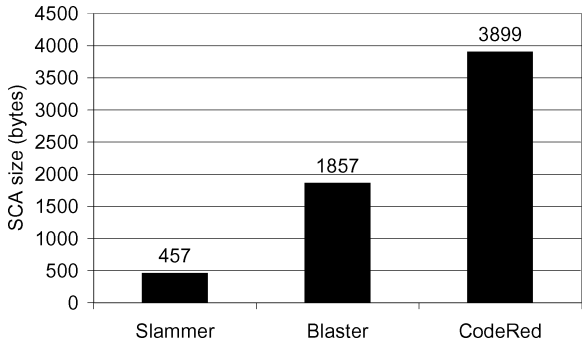

Fig. 20. SCA sizes in bytes for real worms.

Figure 19 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 milliseconds for Slammer, 3.9 milliseconds for Blaster, and 204.7 milliseconds 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 milliseconds and it takes just over 2.6 seconds 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 20 shows the SCA size in bytes for each worm. The SCAs include a fixed header of 81 bytes that encodes the SCA type, vulnerable program identification, and verification information. The size of the SCAs is small and mostly determined by the size of the worm probe messages.

5.4 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

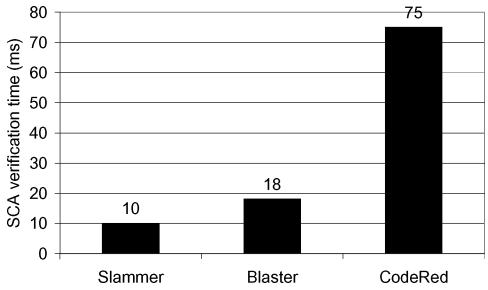

Fig. 21. SCA verification time in milliseconds for real worms.

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 21 shows the average time in milliseconds to verify each SCA. The results are the average of five runs. The standard deviation is 0.5 milliseconds for Slammer, 1.5 milliseconds for Blaster, and 6.5 milliseconds 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 programs 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 programs [Vrable et al. 2005; Fraser and Chang 2003] should enable creation of VMs on demand with low delay.

5.5 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.

5.5.1 *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 Zegura et al. [1996]. 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 millisecond.

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 superpeers: 1,000 of the 500,000 hosts are randomly selected to be superpeers that form a secure Pastry overlay and each ordinary host connects to two superpeers. Each superpeer is able to verify the SCA and is neither vulnerable nor a detector.

We model worm propagation using the epidemic model described in Hethcote [2000] 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 modeled by the following equations.

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

$$\frac{dI_t}{dt} = \beta\, I_t \left(1 - p - \frac{I_t}{S}\right) \tag{2}$$

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 Eqs. (1) and (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 both 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 [Georgatos et al. 2001]. 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 aforesaid 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 superpeers can verify SCAs but other hosts cannot. Unless otherwise stated,

Table III. Simulation Parameters for Modeling Containment of Real Worms

	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

we assume 10 initially infected hosts. Each data point presented is the mean value of 250 runs with an error bar up to the $90th$ 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 neighbors 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 modeling worm outbreaks and countermeasures is still an area of active research [Moore et al. 2003; Zou et al. 2003; Chen et al. 2003; Vojnović and Ganesh 2005; Ganesh et al. 2006], the preceding worm-spreading model has been shown to describe accurately outbreaks of real worms [Moore et al. 2003], and we parameterized the model with measurements from our implementation of Vigilante and with data collected during real outbreaks.

5.5.2 *Containment of Real Worms and Beyond.* First, we evaluate the effectiveness of Vigilante with Slammer, CodeRed, and Blaster. Table III lists the parameter settings used for each worm. The infection rates (β) and susceptible population (S) for Slammer and CodeRed are based on observed behavior reported by Moore et al. [2003]. 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 5.3 and 5.4.

Figure 22 shows the infected percentage (i.e., 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 23 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 Figure 23(b); for example, with a worm infection rate of 8β, the number of infected hosts doubles approximately every second. Because the initially

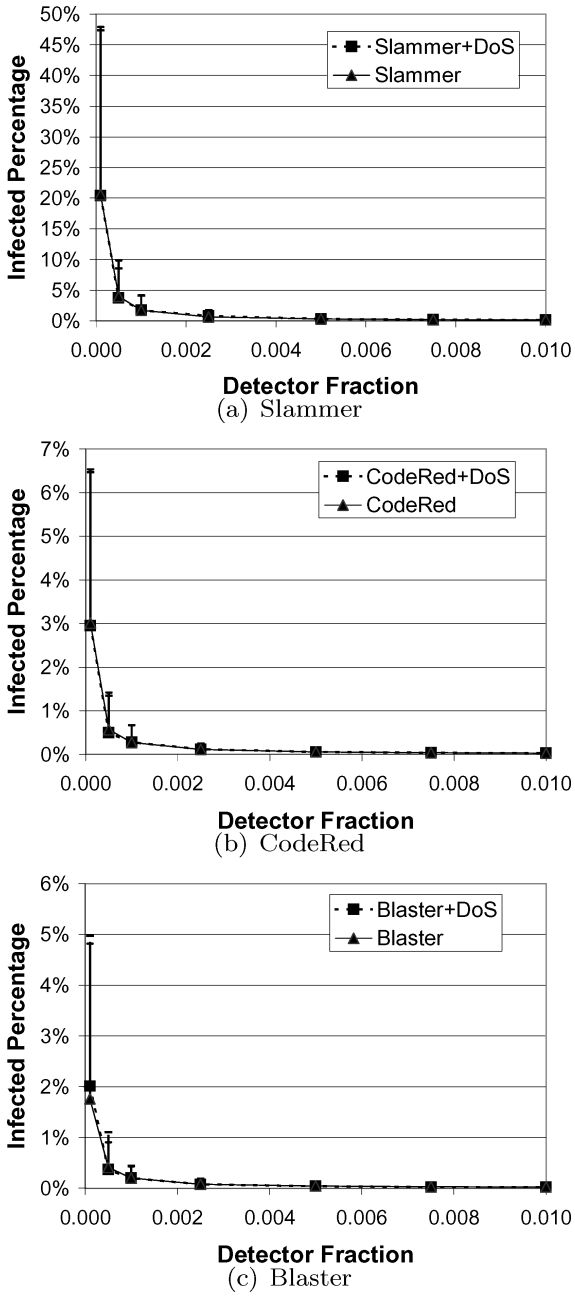

Fig. 22. Containment of Slammer, CodeRed, and Blaster using parameter settings in Table III, both with and without DoS attacks. Each data point is the mean value with an error bar up to the $90th$-percentile value.

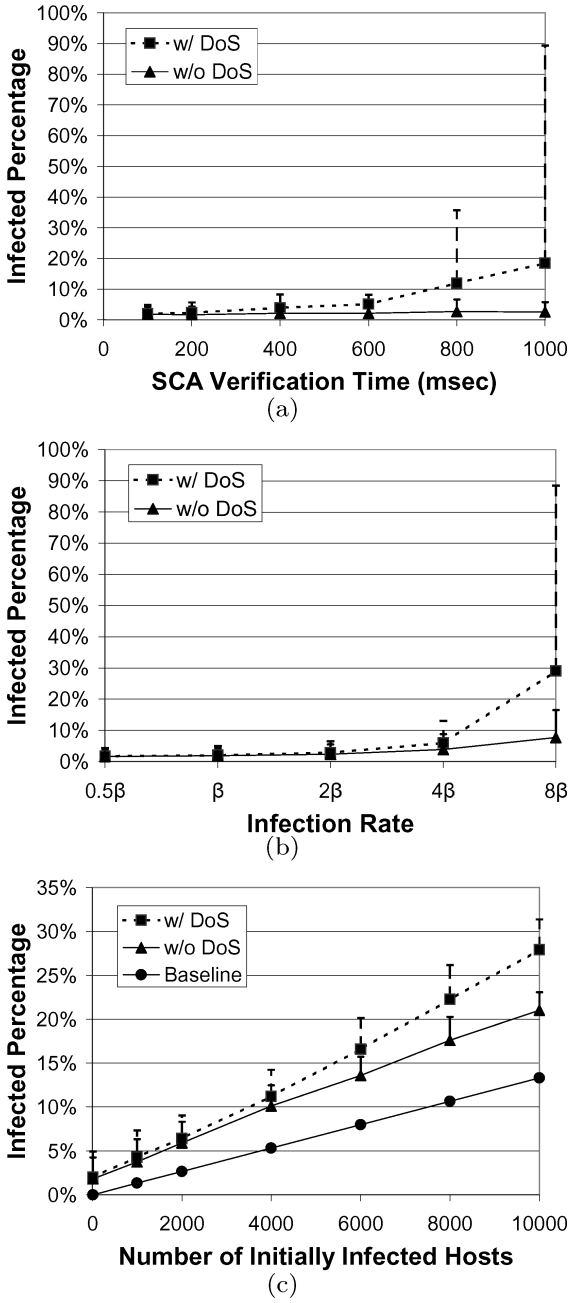

Fig. 23. The effect of SCA verification time, infection rate, and number of initially infected hosts, both with and without DoS attacks. Each data point is the mean value with an error bar up to the $90th$-percentile value.

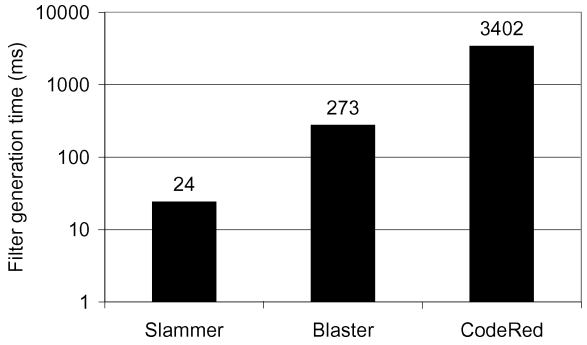

Fig. 24. Filter-generation time for real worms.

infected hosts are counted in the infected percentages reported, the baseline in Figure 23(c) shows the contribution of 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 milliseconds, 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 an SCA verification time of 1000 milliseconds, with infection rate of 8β, or with 10,000 initially infected nodes. Do note that these settings are an order of magnitude worse than the worst of real worms.

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 becomes increasingly vulnerable to DoS attacks as the verification time increases.

Other attacks on the distribution of SCAs have also been analyzed in recent work: Raiciu et al. [2006] 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.

5.6 Protection

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

5.6.1 *Filter Generation.* The first experiment measures the time to generate a filter from an SCA that has already been verified. Figure 24 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 milliseconds for Slammer, 5.1 milliseconds for Blaster, and 205.3 milliseconds 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 rewriting tool needs to dynamically translate code for a number of libraries that are loaded on demand.

```
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
```

Fig. 25. IA-32 code for a condition of Slammer's filter.

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 nonzero (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 condition. For example, in Slammer, the condition that the first byte is equal to 0x04 is computed by the code in Figure 25. 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 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 these 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 make 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 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 [Ptacek and Newsham 1998].

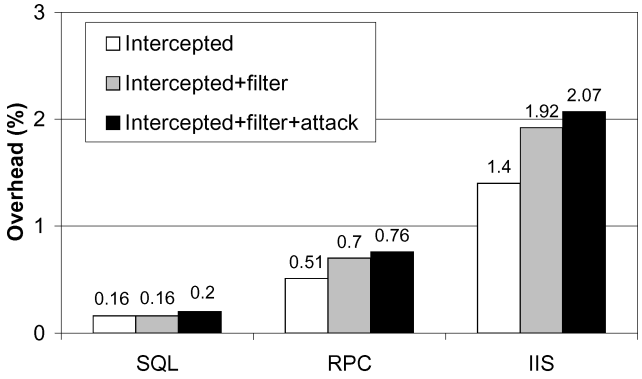

Fig. 26. CPU overhead of network traffic interception and filter deployment.

5.6.2 *Overhead of Deployed Filters.* We also measured the performance overhead introduced by deployed filters. Filters were deployed by binary rewriting the vulnerable programs. We used Detours [Hunt and Brubacher 1999] to intercept calls to the socket interface and install the filters immediately above functions that receive data.

We ran three experiments for each vulnerable program and measured the overhead with a sampling profiler. The first experiment (*intercepted*) ran the program with just the socket interface being intercepted. The second experiment (*intercepted* + *filter*) ran the program 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 program 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 next. Figure 26 shows the overhead for each of the experiments for the three vulnerable programs. The results are the average of five runs. The overhead is low in all cases.

SQL. For Slammer the vulnerable program is SQL Server. We generated load using the TPC-C benchmark [TPC 1999] 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 26 shows that the CPU consumed by the interception is just 0.16%. When Slammer filters are then 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 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%.

RPC. For Blaster the vulnerable program is Microsoft Windows RPC service. We generated a custom workload using requests to look-up and register an RPC

interface. We loaded the RPC service using 3 client hosts that look-up the RPC interface and 1 local client that registers the interface. Figure 26 shows that 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 look-up the interface.

IIS. For CodeRed the vulnerable program is Microsoft IIS Server. We generated a workload using the requests from the SpecWeb99 [SPEC 2008] 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 26 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%.

5.7 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 superpeers. 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 superpeers to a host running the vulnerable program. This provides approximately the same number of hops as the simulations described in Section 5.5.

We measured the time in milliseconds from the moment the worm probe reached the detector until 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 program 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 milliseconds for Slammer, 305 milliseconds for Blaster, and 3044 milliseconds for CodeRed. The results are the average of five runs. The standard deviation is 12.2 milliseconds for Slammer, 9.0 milliseconds for Blaster, and 202.0 milliseconds 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 restarting the vulnerable program. 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.

6. 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 section discusses previous proposals in each of these areas.

6.1 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 hosts 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, namely vulnerable programs, is not visible at the network level.

6.1.1 *Firewalls.* Firewalls [Cheswick et al. 2003] comprise 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 interaction 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 [CERT 2001]. Worms can also exploit virtual private network connections and infected laptops to penetrate enterprise networks. After infecting one host 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 hosts on enterprise networks, they do not provide a general solution for containment.

Personal firewalls, that is, 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.

6.1.2 *Address Blacklisting.* Several systems are based on the idea of blocking network traffic from infected hosts, thus preventing them from infecting other hosts. Early proposals identified infected hosts by analyzing host connectivity graphs [Staniford-Chen et al. 1996]. 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 hosts by detecting scanning behavior. Mirage networks [Mirage 2006] and Forescout [Forescout 2006] mark hosts as infected

if they send messages to unallocated (dark) IP addresses. Worms can avoid this type of detector by not using dark IP addresses. The systems in Ganger et al. [2002] and Whyte et al. [2005] consider hosts infected if they use IP addresses without first resolving the corresponding DNS [Mockapetris 1987] 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 [Toth and Kruegel 2002b; Jung et al. 2004; Schechter et al. 2004; Weaver et al. 2004] because they try to contact unreachable addresses. Jung et al. [2004] 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. [2004] proposed a simplification of TRW that uses a threshold on an estimate of the difference between the number of failed and successful connections. Bro [Paxson 1999] uses a configurable threshold on the number of failed connections. Snort [Roesch 1999] and Network Security Monitor [Heberlein et al. 1990] do not look at failed connections; instead, they monitor the rate at which unique destination addresses are contacted. If hosts exceed a threshold of new addresses contacted in a given interval, they can be flagged as infected. Finally, SPICE [Staniford et al. 2002] is an algorithm to detect 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 [2004] and Ganesh et al. [2006] analyze the conditions under which scanning detection and subsequent blacklisting can provide containment. Staniford [2004] discusses the importance of an "epidemic threshold" for these systems: If on average an infected host can find more than one victim before being blacklisted, the number of infected hosts will still grow exponentially. Weaver et al. [2004] argue 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 (e.g., topological worms that exploit information about hosts in infected hosts in order to propagate, thus avoiding scanning). False positives are another problem for these systems because several normal network services exhibit scanning-like behavior [Jung 2006]. 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.

6.1.3 *Throttling Connections.* A variant of blacklisting is throttling: limiting the resources used by infected hosts without blocking all traffic from those hosts. Williamnson [2002] proposed limiting the rate of connections to new addresses. This can be seen as a form of delaying execution at infected hosts, as proposed by Somayaji and Forrest [2000]. This approach limits the impact of

false positives by allowing the hosts to continue active, albeit with degraded performance. On the other hand, this only slows the spread of worms, without providing containment.

6.1.4 *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. [2003] showed it 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 hosts 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 and Arnold [1994] 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.[4] The candidate signatures are then filtered to minimize the probability of false positives.

More recently, Honeycomb [Kreibich and Crowcroft 2003] proposed generating byte-string signatures from the traffic observed at honeypots. Honeycomb assumes all traffic received by 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 [Kim and Karp 2004] 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 host generating such connection requests is scanning for vulnerable hosts. 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.

[4]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.

The system is also configured with a minimum signature size, which can result in false negatives, especially with polymorphic worms.

Earlybird [Singh et al. 2004] 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 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 [Wang et al. 2005] 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 [Fogla et al. 2006] showed that it can be evaded.

Polygraph [Newsome et al. 2005] 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 [Perdisci et al. 2006] shows that attacks that generate fake anomalous network flows can prevent Polygraph from reliably generating useful signatures.

PADS [Tang and Chen 2005] generates signatures comprised of a sequence of byte frequency distributions. The authors show that PADS works for some cases, but it is unclear whether 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 nonworm traffic from the signature generation procedure, but the worm can still generate bogus traffic after infecting a host.

Nemean [Yegneswaran et al. 2005] 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 [2002a] 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 can 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 [Chinchani and van den Berg 2005; Kruegel et al. 2005; Wang et al. 2006]. Their system assumes attack messages will have a relatively long region with instructions that have no effect (sometimes called a NOP *sledge* [Toth and Kruegel 2002a]), since this is a 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 attack, several systems [Chinchani and van den Berg 2005; Kruegel et al. 2005; Wang et al. 2006] 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.

Techniques that identify code in messages are more resilient against 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 [Wang et al. 2006]) 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 [rix@hert.org 2001]; for instance, the systems in Toth and Kruegel [2002a] and Wang et al. [2006] use protocol-specific information to decode the network messages, before trying to find executable code.

6.2 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.

6.2.1 *Avoid/Remove Defects.* Type-safe languages [Cardelli 2004; Madhavapeddy 2006] 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 attack [SecurityFocus

2002]. Finally, there is a 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 [Necula et al. 2002] and Cyclone [Jim et al. 2002] 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 nontrivial 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 [Boyer et al. 1975] and Lint [Johnson 1984] were some of the early tools in this space. More recently, several tools [Bush et al. 2000; Yang et al. 2004; Xie and Aiken 2005] have been used to find defects in large programs. Some tools have been specifically designed to find security vulnerabilities [Wagner et al. 2000; Evans and Larochelle 2002; Johnson and Wagner 2004; Shankar et al. 2001; Larochelle and Evans 2001; Avots et al. 2005; Livshits and Lam 2005].

Most of these tools can generate false positives, that is, they report defects which are not real. One reason 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 behavior [Engler et al. 2001].

6.2.2 *Detect/Prevent Exploits.* Since static tools can have false positives and 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 consisted of identifing anomalous patterns of system calls [Forrest et al. 1996]. Wagner and Soto [2002] showed that *mimicry attacks* can elude this type of detection, and Kruegel et al. [2005] showed how to automate these attacks, even for recent improvements on the original technique [Feng et al. 2003; Giffin et al. 2004; Sekar et al. 2001].

Other early systems protected specific control data structures, such as return addresses. StackGuard [Cowan et al. 1998] 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 method detects attacks that overflow buffers on the stack because the overflow overwrites the canary value on the way to overwriting the return address. StackShield [Vendicator 2001], RAD [Chiueh and Hsu 2001], and Libverify [Baratloo et al. 2000] proposed keeping copies of return addresses separate from the normal stack. This allows them to detect overwrites of return addresses by comparing the saved values with values on the normal stack.

They can also recover the original return addresses. Libsafe [Baratloo et al. 2000] provided implementations of C library functions that do additional bound checks to avoid overwriting return addresses. FormatGuard [Cowan et al. 2001] provides safe implementations of C library functions that use format strings. PointGuard [Cowan et al. 2003] 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 [Wilander and Kamkar 2003; Bulba and Kil3r 2000].

More recently, DIRA proposed protecting all control data structures [Smirnov and Chiueh 2005] 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 [Wilander and Kamkar 2003; Chiueh and Hsu 2001].

Backwards-compatible bounds checking for C [Jones and Kelly 1997] 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 [Ruwase and Lam 2004] 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 Jones and Kelly [1997] 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) [Ruwase and Lam 2004], yielding a slowdown of up to 2.3X, but this increases the number of false negatives.

Program shepherding [Kiriansky et al. 2002] 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 rewriter [Bruening et al. 2001] to monitor the program's execution and to ensure that every control-flow transition is allowed by the control-flow graph. Control-Flow Integrity [Abadi et al. 2005] checks that control-flow transitions follow the computed control-flow graph with inlined checks based on a static binary rewriter.

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 [Chen et al. 2005]. 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 [Costa et al. 2005, 2004], three systems [Suh et al. 2004; Crandall and Chong 2004; Newsome and Song 2005] have proposed similar mechanisms for detection, that do not require access to source code. The idea of securing information flows can be traced back to the work of Fenton [1974a, 1973] and Denning [1976]. Recently these ideas have also been explored in the context of programming languages [Myers 1999] and distributed systems [Zheng et al. 2003]; in particular, Perl taint mode [PERL 2006] proposed tracking input data and preventing unsafe uses of that data in the context of a scripting language, and Chow et al. [2004] proposed tracking the lifetime of sensitive information, such as passwords, through memory and CPU registers. More recently, Suh et al. [2004] 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 multigranularity 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; and 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; and 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 [Crandall and Chong 2004] is a hardware microarchitecture that implements Biba's low-watermark policy [Biba 1977]. 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 those attacks that hijack control flow by overwriting control data structures. Vigilante also detects attacks that corrupt noncontrol data used in system calls, as well as attacks that redirect execution to dirty memory regions, without corrupting control data structures.

TaintCheck [Newsome and Song 2005] tracks input data by instrumenting binaries using Valgrind [Nethercote and Seward 2003]. 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 with Minos, it does not check whether 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 with 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 are observed can be recorded to avoid raising security traps there.

The work in Chen et al. [2005] evaluated a security policy that generates security traps when memory writes use dirty pointers. This policy had been proposed in Suh et al. [2004], but had not been evaluated in the context of noncontrol-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 and Chong [2004] discussed 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 positive. If a detector generates an alert that cannot be verified, it is simply discarded.

Since its original publication [Costa et al. 2005, 2004], the dynamic data-flow analysis algorithm has also be used by several systems. Ho et al. [2006] proposed an implementation based on the Xen [Barham et al. 2003] virtual machine monitor that automatically transitions from emulation to direct CPU execution when none of the CPU's registers is dirty. Argos [Portokalidis et al. 2006] uses an implementation based on QEMU [2006] 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 Forrest et al. [1997]. Randomizing the memory layout of processes was originally implemented by the PAX [2001] 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 [Bhatkar et al. 2005, 2003; Xu et al. 2003]. Several attacks against address randomization have been proposed [Durden 2002]. It has been shown that for some implementations it is possible to discover the addresses of relevant objects by brute-force attacks [Shacham et al. 2004]. Information leakage attacks [Shacham et al. 2004] 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 [Kc et al. 2003; Barrantes et al. 2003]. The idea is to create process-specific randomized instruction sets by using a simple encoding of instructions, for example, 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 Sovarel et al. [2005].

Finally, it is important to note that the diversity of the proposed detection mechanisms 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 hosts in the system.

6.2.3 *Survivability.* Several systems have proposed mechanisms that, like Vigilante filters, allow vulnerable programs to continue execution while being attacked.

Rinard et al. [2004] proposed *failure-oblivious computing*. They use a C compiler that inserts runtime checks for illegal memory accesses using the C range error detector [Ruwase and Lam 2004]. 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 [Smirnov and Chiueh 2005] is a compiler extension that can log updates to memory, and allows rolling back vulnerable programs to the entry point of a function. Sidiroglou et al. [2005] 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 [Qin et al. 2005] 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 has been 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 are committed is problematic; for instance, the 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 the execution environment for a threshold interval, but it still discards them after this 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 less likely to affect the correct execution of the protected programs.

Sidiroglou et al. [2005] 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 this 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 [Pasupulati et al. 2004] 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 [Hsu and Chiueh 2004] and TaintCheck [Newsome and Song 2005] 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 corresponding to instructions that transfer control to the worm code [Crandall et al. 2005].

ARBOR [Liang and Sekar 2005a] 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 [Liang and Sekar 2005b] 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. In contrast, Vigilante does not require network protocol information. Moreover, COVERS does not provide guarantees on the rate of false positives or false negatives.

Several systems provide interesting alternatives to deploy Vigilante filters. IntroVirt [Joshi et al. 2005] 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 whether vulnerabilities were exploited in the past. We could deploy Vigilante filters as IntroVirt predicates. Shield [Wang et al. 2004] uses host-based filters to block vulnerabilities, but these filters are generated manually. We could use Shield's infrastructure to deploy our filters.

6.3 Artificial Immune Systems

Several projects have contributed to the design of artificial immune systems. Cohen [1987] studied computer viruses, and Kephart et al. [1997] 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 Kephart et al. [1997]. Hofmeyr and Forrest [2000] describe 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 [Forrest et al. 1997]. Interestingly, the argument that monocultures contribute to improved security has also been made [Locasto et al. 2006].

Several authors have proposed theoretical models for predicting characteristics of worm epidemics and for analyzing immunization systems [Kephart and White 1991; Wang et al. 2000; Staniford et al. 2004, 2002; Moore et al. 2003; Zou et al. 2003; Chen et al. 2003; Staniford et al. 2004; Vojnović and Ganesh 2005; Goldenberg et al. 2005; Ganesh et al. 2006]. 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 hosts can safely share information about the attacks in a timely manner, and how hosts can protect themselves efficiently.

7. CONCLUSIONS

7.1 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 spread 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, namely 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 vulnerability condition slicing: 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.

7.2 Future Work

There are several promising directions for future work. Recently, we proposed techniques that can prevent both control- and noncontrol-data attacks by enforcing safety properties that we call *data-flow integrity* [Castro et al. 2006] and *write integrity* [Akritidis et al. 2008]. We could integrate these techniques as new detectors 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 [Costa et al. 2007], and our work also inspired others to follow this direction [Brumley et al. 2006]. Analyzing more execution paths (besides the path identified by an SCA) and using more static analysis techniques 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* [Necula and Lee 1996], where logic proofs of vulnerability are exchanged by hosts.

Finally, it is interesting to consider integrating Vigilante with network telescopes Moore et al. [2004, 2001] and honeyfarms [Vrable et al. 2005]. By redirecting 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.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. We thank J.-P. Martin and S. Hand for many discussions and comments. We thank S. Bhansali and D. Mihocka for their help with Nirvana. We thank E. Traut for discussions about Virtual PC. We thank R. Wilhelm and H. Uijterwaal for access to the RIPE data. We thank J. Gorm Hansen for implementing the Windows NX detector. We thank M. Peinado, L. Visconti, and J. Manferdelli for many discussions about this project and for taking it to the next level. We thank A. Herbert for his comments and support for this work.

REFERENCES

ABADI, M., BUDIU, M., ERLINGSSON, U., AND LIGATTI, J. 2005. Control-Flow Integrity: Principles, implementations, and applications. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*.

AKAMAI. 2000. Press release: Akamai helps mcafee.com support flash crowds from iloveyou virus.

AKRITIDIS, P., CADAR, C., RAICIU, C., COSTA, M., AND CASTRO, M. 2008. Preventing memory error exploits with WIT. In *Proceedings of the IEEE Symposium on Security and Privacy*.

AVOTS, D., DALTON, M., LIVSHITS, V. B., AND LAM, M. S. 2005. Improving software security with a C pointer analysis. In *Proceedings of the 27th International Conference on Software Engineering*.

BAILEY, M., COOKE, E., JAHANIAN, F., WATSON, D., AND NAZARIO, J. 2005. The Blaster worm: Then and now. *IEEE Secur. Privacy 3,* 4, 26–31.

BARATLOO, A., SINGH, N., AND TSAI, T. 2000. Transparent runtime defense against stack smashing attacks. In *Proceedings of the USENIX Technical Conference*.

BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. 2003. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*.

BARRANTES, E. G., ACKLEY, D. H., PALMER, T. S., STEFANOVIC, D., AND ZOV, D. D. 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*.

BETHENCOURT, J., FRANKLIN, J., AND VERNON, M. 2005. Mapping Internet sensors with probe response attacks. In *Proceedings of 14th USENIX Security Symposium*.

BHANSALI, S., CHEN, W.-K., DE JONG, S., EDWARDS, A., MURRAY, R., DRINIC, M., MIHOCKA, D., AND CHAU, J. 2006. Framework for instruction-level tracing and analysis of program executuions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*.

BHATKAR, S., DUVARNEY, D. C., AND SEKAR, R. 2003. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of 12th USENIX Security Symposium*.

BHATKAR, S., SEKAR, R., AND DUVARNEY, D. C. 2005. Efficient techniques for comprehensive protection from memory error exploits. In *Proceedings of 14th USENIX Security Symposium*.

BIBA, K. J. 1977. Integrity considerations for secure computer systems. Tech. Rep. TR-3153, MITRE. April.

BLEXIM. 2002. Basic integer overflows. *Phrack* 60.

BOCHS. 2006. Bochs ia-32 emulator. http://bochs.sourceforge.net.

BOYER, R. S., ELSPAS, B., AND LEVITT, K. N. 1975. SELECT—A formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*.

BRUENING, D., DUESTERWALD, E., AND AMARASINGHE, S. 2001. Design and implementation of a dynamic optimization framework for Windows. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*.

BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. 2006. Towards automatic generation of vulnerability signatures. In *Proceedings of the IEEE Symposium on Security and Privacy*.

BULBA AND KIL3R. 2000. Bypassing stackguard and stackshield. *Phrack 10,* 46 (May).

BUSH, W. R., PINCUS, J. D., AND SIELAFF, D. J. 2000. A static analyzer for finding dynamic programming errors. *Softw. Practice Exper. 30*, 775–802.

CARDELLI, L. 2004. Type systems. In *The Computer Science and Engineering Handbook.* CRC Press.

CASTRO, M., COSTA, M., AND HARRIS, T. 2006. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*.

CASTRO, M., COSTA, M., AND ROWSTRON, A. 2004. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of the International Conference on Dependable Systems and Networks*.

CASTRO, M., DRUSCHEL, P., GANESH, A., ROWSTRON, A., AND WALLACH, D. S. 2002. Security for structured peer-to-peer overlay networks. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*.

CERT. 2001. Cert advisory ca-2001-26 nimda worm. http://www.cert.org/advisories/ca-2001-26.html.

CERT. 2005. Technical cyber security alerts. http://www.us-cert.gov.

CHEN, S., XU, J., NAKKA, N., KALBARCZYK, Z., AND IYER, R. K. 2005. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the International Conference on Dependable Systems and Networks*.

CHEN, S., XU, J., SEZER, E. C., GAURIAR, P., AND IYER, R. K. 2005. Non-Control-Data attacks are realistic threats. In *Proceedings of 14th USENIX Security Symposium*.

CHEN, Z., GAO, L., AND KWIAT, K. 2003. Modelling the spread of active worms. In *Proceedings of the 22th IEEE Conference on Computer Communications*.

CHESWICK, W. R., BELLOVIN, S. M., AND RUBIN, A. D. 2003. *Firewalls and Internet Security: Repelling the Wily Hacker.* Addison-Wesley.

CHINCHANI, R. AND VAN DEN BERG, E. 2005. A fast static analysis approach to detect exploit code inside network flows. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*.

CHIUEH, T. AND HSU, F. 2001. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*.

CHOW, J., PFAFF, B., GARFINKEL, T., CHRISTOPHER, K., AND ROSENBLUM, M. 2004. Understanding data lifetime via whole system simulation. In *Proceedings of 13th USENIX Security Symposium*.

COHEN, F. 1987. Computer viruses, theory and experiments. *Comput. Secur. 6*, 22–35.

CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Electrical Engineering and Computer Science Series. MIT Press.

COSTA, M., CASTRO, M., ZHOU, L., ZHANG, L., AND PEINADO, M. 2007. Bouncer: Securing software by blocking bad input. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*.

COSTA, M., CROWCROFT, J., CASTRO, M., AND ROWSTRON, A. 2004. Can we contain Internet worms? In *Proceedings of the 3rd Workshop on Hot Topics in Networks*.

COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. 2005. Vigilante: End-to-End containment of Internet worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*.

COWAN, C., BARRINGER, M., BEATTIE, S., KROAH-HARTMAN, G., FRANTZEN, M., AND LOKIER, J. 2001. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*.

COWAN, C., BEATTIE, S., JOHANSEN, J., AND WAGLE, P. 2003. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*.

COWAN, C., PU, C., MAIER, D., HINTON, H., WADPOLE, J., BAKKE, P., BEATTIE, S., GRIER, A., WAGLE, P., AND ZHANG, Q. 1998. Stackguard: Automatic detection and prevention of buffer-overrun attacks. In *Proceedings of the 7th USENIX Security Symposium*.

CRANDALL, J. R. AND CHONG, F. T. 2004. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*.

CRANDALL, J. R., SU, Z., WU, S. F., AND CHONG, F. T. 2005. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*.

DARK SPYRIT. 1999. Win32 buffer overflows. *Phrack 9,* 55.

DENNING, D. 1976. A lattice model of secure information flow. *ACM Trans. Commun. 19*, 5, 236–243.

DIJKSTRA, E. W. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM 18*, 8 (Aug.), 453–457.

DOUCEUR, J. R. 2002. The Sybil attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*.

DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. 2002. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation*.

DURDEN, T. 2002. Bypassing pax aslr protection. *Phrack* 59 (Jul.).

EICHIN, M. W. AND ROCHLIS, J. A. 1989. With microscope and tweezers: An analysis of the Internet virus of November 1988. In *Proceedings of the IEEE Symposium on Security and Privacy*.

ELNOZAHY, E. N., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv. 34*, 3 (Sept.), 375–408.

ENGLER, D., CHEN, D. Y., HALLEM, S., CHOU, A., AND CHELF, B. 2001. Bugs as deviant behaviour: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*.

EVANS, D. AND LAROCHELLE, D. 2002. Improving security using extensible lightweight static analysis. *IEEE Softw.*

FENG, H., KOLESNIKOV, O., FOGLA, P., LEE, W., AND GONG, W. 2003. Anomaly detection using system call information. In *Proceedings of the IEEE Symposium on Security and Privacy*.

FENTON, J. 1973. Information protection systems. Ph.D. thesis, University of Cambridge.

FENTON, J. 1974a. An abstract computer model demonstrating directional information flow. University of Cambridge, Cambridge, UK.

FENTON, J. S. 1974b. Memoryless subsystems. *Comput. J. 17*, 2, 143–147.

FOGLA, P., SHARIF, M., PERDISCI, R., KOLESNIKOV, O., AND LEE, W. 2006. Polymorphic blending attacks. In *Proceedings of 15th USENIX Security Symposium*.

FORESCOUT. 2006. Wormscout. http://www.forescout.com/wormscout.html.

FORREST, S., HOFMEYR, S. A., SOMAYAJI, A., AND LONGSTAFF, T. A. 1996. A sense of self for Unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*.

FORREST, S., SOMAYAJI, A., AND ACKLEY, D. 1997. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*.

FRASER, K. AND CHANG, F. 2003. Operating System I/O Speculation: How two invocations are faster than one. In *Proceedings of the USENIX Annual Technical Conference*.

GANESH, A., GUNAWARDENA, D., KEY, P., MASSOULIE, L., AND SCOTT, J. 2006. Efficient quarantining of scanning worms: Optimal detection and coordination. In *Proceedings of the 25th IEEE Conference on Computer Communications*.

GANGER, G., ECONOMU, G., AND BIELSKI, S. 2002. Self-Securing network interfaces: What, why and how. Tech. Rep. CS-02-144, Carnegie Mellon University. May.

GEORGATOS, F., GRUBER, F., KARRENBERG, D., SANTCROOS, M., UIJTERWAAL, H., AND WILHELM, R. 2001. Providing Active Measurements as a Regular Service for ISPs. http://www.ripe.net/ttm.

GERA AND RIQ. 2002. Advances in format string exploitation. *Phrack* 59 (Jul.).

GIFFIN, J., JHA, S., AND MILLER, B. P. 2004. Efficient context-sensitive intrusion detection. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*.

GOLDENBERG, J., SHAVITT, Y., SHIR, E., AND SOLOMON, S. 2005. Distributive immunization of networks against viruses using the 'honey pot' architecture. *Nature Phys. 1*, 184–188.

HEBERLEIN, L. T., DIAS, G., K, L., WOOD, B. M. J., AND WOLBER, D. 1990. A network security monitor. In *Proceedings of the IEEE Symposium on Security and Privacy*.

HETHCOTE, H. W. 2000. The mathematics of infectious deseases. *SIAM Rev. 42*, 4, 599–653.

HO, A., FETTERMAN, M., CLARK, C., WARFIELD, A., AND HAND, S. 2006. Practical taint-based protection using demand emulation. In *Proceedings of the SiGOPS European Conference on Computer Systems (EuroSys)*.

HOFMEYR, S. A. AND FORREST, S. 2000. Architecture for an artificial immune system. *Evolutionary Comput. 8*, 4(Dec.), 443–473.

HOLZ, T. AND RAYNAL, F. 2005. Detecting honeypots and other suspicious environments. In *Workshop on Information Assurance and Security*.

HSU, F. AND CHIUEH, T. 2004. CTCP: A centralized TCP architecture for networking security. In *Proceedings of the Annual Computer Society Applications Conference (ACSAC)*.

HUA, W., OHLUND, J., AND BUTTERKLEE, B. 1999. Unraveling the mysteries of writing a winsock 2 layered service provider. *Microsoft Syst. J.*

HUNT, G. AND BRUBACHER, D. 1999. Detours: Binary interception of Win32 functions. In *USENIX Windows NT Symposium*.

INTEL. 1999. Intel architecture software developer's manual, vol. 2: Instruction set reference.

JIM, T., MORRISETT, G., GROSSMAN, D., HICKS, M., CHENEY, J., AND WANG, Y. 2002. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*.

JOHNSON, R. AND WAGNER, D. 2004. Finding user/kernel pointer bugs with type inference. In *Proceedings of 13th USENIX Security Symposium*.

JOHNSON, S. C. 1984. Lint, a C program checker. In *Unix Programmer's Manual, 4.2*. Berkeley Software Distribution Supplementary Documents.

JONES, R. AND KELLY, P. 1997. Backwards-Compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the International Workshop on Automatic Debugging*.

JOSHI, A., KING, S., DUNLAP, G., AND CHEN, P. 2005. Detecting past and present intrusions through vulnerability-specific predicates. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*.

JP. 2003. Advanced doug lea's malloc exploits. *Phrack* 61 (Sept.).

JUNG, J. 2006. Real-Time detection of malicious network activity using stochastic models. Ph.D. thesis, Massachusetts Institute of Technology.

JUNG, J., PAXSON, V., BERGER, A. W., AND BALAKRISHNAN, H. 2004. Fast portscan detection using sequential hypothesis testing. In *Proceedings of the IEEE Symposium on Security and Privacy*.

KC, G. S., KEROMYTIS, A. D., AND PREVELAKIS, V. 2003. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*.

KEPHART, J. O. AND ARNOLD, W. C. 1994. Automatic extraction of computer virus signatures. In *International Virus Bulletin Conference*.

KEPHART, J. O., SORKIN, G. B., SWIMMER, M., AND WHITE, S. R. 1997. Blueprint for a computer immune system. In *International Virus Bulletin Conference*.

KEPHART, J. O. AND WHITE, S. R. 1991. Directed-Graph epidemiological models of computer viruses. In *Proceedings of the IEEE Symposium on Security and Privacy*.

KIM, H. AND KARP, B. 2004. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*.

KING, J. C. 1976. Symbolic execution and program testing. *Commun. ACM 19*, 7 (Jul.), 385–394.

KIRIANSKY, V., BRUENING, D., AND AMARASINGHE, S. P. 2002. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*.

KREIBICH, C. AND CROWCROFT, J. 2003. Honeycomb Creating intrusion detection signatures using honeypots. In *Proceedings of the 2nd Workshop on Hot Topics in Networks*.

KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. 2005. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*.

KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSONAND, W., AND VIGNA, G. 2005. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium*.

LAROCHELLE, D. AND EVANS, D. 2001. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*.

LIANG, Z. AND SEKAR, R. 2005a. Automatic generation of buffer overflow signatures: An approach based on program behavior models. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

LIANG, Z. AND SEKAR, R. 2005b. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*.

LIVSHITS, V. B. AND LAM, M. S. 2005. Finding security vulnerabilities in java applications using static analysis. In *Proceedings of the 14th USENIX Security Symposium*.

LOCASTO, M., SIDIROGLOU, S., AND KEROMYTIS, A. 2006. Software self-healling using collaborative application communities. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium*.

MADHAVAPEDDY, A. 2006. Creating high-performance statically type-safe network applications. Ph.D. thesis, University of Cambridge.

MIRAGE. 2006. Mirage networks. http://www.miragenetworks.com.

MOCKAPETRIS, P. 1987. Domain names: Concepts and facilities. Tech. Rep. RFC-1034, Internet Engineering Task Force. November.

MOORE, D., PAXSON, V., SAVAGE, S., SHANNON, C., STANIFORD, S., AND WEAVER, N. 2003. Inside the Slammer worm. *IEEE Secur. Privacy 1*, 4 (Jul.).

MOORE, D., SHANNON, C., AND BROWN, J. 2002. Code-Red: A case study on the spread and victims of an Internet worm. In *ACM Internet Measurement Workshop*.

MOORE, D., SHANNON, C., VOELKER, G., AND SAVAGE, S. 2003. Internet quarantine: Requirements for containing self-propagating code. In *Proceedings of the 22th IEEE Conference on Computer Communications*.

MOORE, D., SHANNON, C., VOELKER, G. M., AND SAVAGE, S. 2004. Network telescopes: Tech. Rep. CS2004-0795, University of California at San Diego. July.

MOORE, D., VOELKER, G. M., AND SAVAGE, S. 2001. Inferring Internet denial of service activity. In *Proceedings of the 10th USENIX Security Symposium*.

MYERS, A. C. 1999. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

NECULA, G. C. AND LEE, P. 1996. Safe kernel extensions without run-time checking. In *Proceedings of the ACM USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

NECULA, G. C., MCPEAK, S., AND WEIMER, W. 2002. CCured: Type-Safe retrofitting of legacy code. In *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

NERGAL. 2001. The advanced return-into-lib(c) exploits: Pax case study. *Phrack 58*.

NETHERCOTE, N. AND SEWARD, J. 2003. Valgrind: A program supervision framework. In *Proceedings of the 3rd Workshop on Runtime Verification (RV)*.

NEWSOME, J., KARP, B., AND SONG, D. 2005. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*.

NEWSOME, J. AND SONG, D. 2005. Dynamic taint analysis for automatic detection, analysis and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*.

ONE, A. 1996. Smashing the stack for fun and profit. *Phrack 7*, 49 (Nov.).

PASUPULATI, A., COIT, J., LEVITT, K., WU, S. F., LI, S. H., KUO, J. C., AND FAN, K. P. 2004. Buttercup: On network-based detection of polymorphic buffer overflow vulnerabilities. In *Proceedings of the IEEE IFIP Network Operations and Management Symposium (NOMS)*.

PAX. 2001. PaX system. http://pax.grsecurity.net/.

PAXSON, V. 1999. Bro. A system for detecting network intruders in real time. *Comput. Netw. 31*, 23-24 (Dec.), 2435–2463.

PERDISCI, R., DAGON, D., LEE, W., FOGLA, P., AND SHARIF, M. 2006. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the IEEE Symposium on Security and Privacy*.

PERL. 2006. Perl security manual page. http://www.perldoc.com.

PORTOKALIDIS, G., SLOWINSKA, A., AND BOS, H. 2006. Argos: An emulator for fingerprinting zero-day attacks. In *Proceedings of the SIGOPS European Conference on Computer Systems (EuroSys)*.

PROVOS, N. 2004. A virtual honeypot framework. In *Proceedings of the 13th USENIX Security Symposium*.

PTACEK, T. H. AND NEWSHAM, T. N. 1998. Insertion, evasion, and denial of service: Eluding network intrusion detection. Tech. Rep., Secure Networks, Inc. January.

QEMU. 2006. Qemu open source processor emulator. http://fabrice.bellard.free.fr/qemu/.

QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. 2005. Rx: Treating bugs as allergies: A safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*.

RAICIU, C., HANDLEY, M., AND ROSENBLUM, D. S. 2006. Exploit hijacking: Side effects of smart defenses. In *Proceedings of the SIGCOMM Workshops*.

RINARD, M., CADAR, C., DUMITRAN, D., ROY, D. M., LEU, T., AND JR., W. S. B. 2004. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*.

RIX@HERT.ORG. 2001. Writing ia32 alphanumeric shellcodes. *Phrack 11*, 57 (Aug.).

ROESCH, M. 1999. Snort: Lightweight intrusion detection for networks. In *Conference on Systems Administration*.

RUWASE, O. AND LAM, M. 2004. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*.

SCHECHTER, S., JUNG, J., AND BERGER, A. 2004. Fast detection of scanning worm infections. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection*.

SECURITYFOCUS. 2002. Microsoft jvm class loader buffer overrun vulnerability. http://www.securityfocus.com/bid/6134.

SEKAR, R., BENDRE, M., DHURJATI, D., AND BOLLINENI, P. 2001. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*.

SHACHAM, H., PAGE, M., PFAFF, B., GOH, E.-J., MODADUGU, N., AND BONEH, D. 2004. On the effectiveness of address space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*.

SHANKAR, U., TALWAR, K., FOSTER, J. S., AND WAGNER, D. 2001. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*.

SHANNON, C. AND MOORE, D. 2004. The spread of the Witty worm. *IEEE Secur. Privacy 2*, 4 (Jul.).

SHINODA, Y., IKAI, K., AND ITOH, M. 2005. Vulnerabilities of passive Internet threat monitors. In *Proceedings of the 14th USENIX Security Symposium*.

SHOCH, J. F. AND HUPP, J. A. 1982. The worm programs: Early experience with a distributed computation. *Commun. ACM 25*, 3 (Mar.), 172–180.

SIDIROGLOU, S., LOCASTO, M. E., BOYD, S. W., AND KEROMYTIS, A. D. 2005. Building a reactive immune system for software services. In *Proceedings of the USENIX Annual Technical Conference*.

SINGH, S., ESTAN, C., VARGHESE, G., AND SAVAGE, S. 2004. Automated worm fingerprinting. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*.

SMIRNOV, A. AND CHIUEH, T. 2005. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*.

SOMAYAJI, A. AND FORREST, S. 2000. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*.

SOVAREL, N., EVANS, D., AND PAUL, N. 2005. Where's the FEEB? The effectiveness of instruction set randomization. In *Proceedings of the 14th USENIX Security Symposium*.

SPAFFORD, E. H. 1989. The Internet worm: Crisis and aftermath. *Commun. ACM 32*, 6 (Jun.), 678–687.

SPEC. Specweb99 benchmark. http://www.spec.org/osg/web99.

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

STANIFORD, S., HOAGLAND, J., AND MCALERNEY, J. 2002. Practical automated detection of stealthy portscans. *J. Comput. Secur. 10*, 105–136.

STANIFORD, S., MOORE, D., PAXSON, V., AND WEAVER, N. 2004. The top speed of flash worms. In *Proceedings of the 2nd Workshop on Rapid Malcode*.

STANIFORD, S., PAXSON, V., AND WEAVER, N. 2002. How to 0wn the Internet in your spare time. In *Proceedings of the 11th USENIX Security Symposium*.

STANIFORD-CHEN, S., CRAWFORD, R., DILGER, M., FRANK, J., HOAGLAND, J., LEVITT, K., AND ZERKLE, D. 1996. GrIDS: A graph-based intrusion detection system for large networks. In *Proceedings of the 19th National Information Systems Security Conference*.

SUH, G. E., LEE, J., AND DEVADAS, S. 2004. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*.

SZOR, P. AND FERRIE, P. 2001. Hunting for metamorphic. In *the International Virus Bulletin Conference*.

TANG, Y. AND CHEN, S. 2005. Defending against Internet worms: A signature-based approach. In *Proceedings of the 24th IEEE Conference on Computer Communications*.

TOTH, T. AND KRUEGEL, C. 2002a. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection*.

TOTH, T. AND KRUEGEL, C. 2002b. Connection-History based anomaly detection. In *the IEEE Information Assurance Workshop*.

TPC. 1999. TPC-C online transaction processing benchmark. http://www.tpc.org/tpcc/default.asp.

VENDICATOR. 2001. Stack shield technical info. http://www.angelfire.com/sk/stackshield.

VOJNOVIĆ, M. AND GANESH, A. 2005. On the race of worms, alerts and patches. In *Proceedings of the 3rd Workshop on Rapid Malcode*.

VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. 2005. Scalability, fidelity and containment in the potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*.

WAGNER, D., FOSTER, J. S., BREWER, E. A., AND AIKEN, A. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 7th Annual Network and Distributed System Security Symposium*.

WAGNER, D. AND SOTO, P. 2002. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*.

WANG, C., KNIGHT, J., AND ELDER, M. 2000. On computer viral infection and the effect of immunization. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

WANG, H. J., GUO, C., SIMON, D. R., AND ZUGENMAIER, A. 2004. Shield: Vulnerability-Driven network filters for preventing known vulnerability exploits. In *Proceedings of the ACM SIGCOMM Data Communications Festival*.

WANG, K., CRETU, G., AND STOLFO, S. J. 2005. Anomalous payload-based worm detection and signature generation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection*.

WANG, X., PAN, C.-C., LIU, P., AND ZHU, S. 2006. Sigfree: A signature-free buffer overflow attack blocker. In *Proceedings of the 15th USENIX Security Symposium*.

WEAVER, N., ELLIS, D., STANIFORD, S., AND PAXSON, V. 2004. Worms vs. perimeters: The case for hard-LANs. In *Hot Interconnects 2*.

WEAVER, N., STANIFORD, S., AND PAXSON, V. 2004. Very fast containment of scanning worms. In *Proceedings of the 13th USENIX Security Symposium*.

WEISER, M. 1984. Program slicing. *IEEE Trans. Softw. Eng. 10*, 4, 352–357.

WHYTE, D., KRANAKIS, E., AND OORSCHOT, P. C. V. 2005. Dns-Based detection of scanning worms in an enterprise network. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*.

WILANDER, J. AND KAMKAR, M. 2003. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*.

WILLIAMNSON, M. M. 2002. Throttling viruses: Restricting propagation to defeat mobile malicious code. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

WINSKEL, G. 1993. *The Formal Semantics of Programming Languages*. MIT Press.

XIE, Y. AND AIKEN, A. 2005. Scalable error detection using Boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

XU, J., KALBARCZYK, Z., AND IYER, R. K. 2003. Transparent runtime randomization for security. In *Proceedings of the IEEE Symposium on Reliability in Distributed Software (SRDS)*.

YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. 2004. Using model checking to find serious file system errors. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*.

YEGNESWARAN, V., GIFFIN, J. T., BARFORD, P., AND JHA, S. 2005. An architecture for generating semantics aware signatures. In *Proceedings of the 14th USENIX Security Symposium*.

ZEGURA, E., CALVERT, K., AND BHATTACHARJEE, S. 1996. How to model an internetwork. In *Proceedings of the Annual Joint Conference of the IEEE Computer Communications Societies (IEEE INFOCOM)*.

ZHENG, L., CHONG, S., MYERS, A. C., AND ZDANCEWIC, S. 2003. Using replication and partitioning to build secure distributed systems. In *Proceedings of the IEEE Symposium on Security and Privacy*.

ZOU, C. C., GAO, L., GONG, W., AND TOWSLEY, D. 2003. Monitoring and early warning for Internet worms. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*.

Received November 2005; revised June 2008; accepted September 2008