Vigilante: End-to-End Containment of Internet Worms

Manuel Costa[1,2], Jon Crowcroft[1], Miguel Castro[2], Antony Rowstron[2],
Lidong Zhou[3], Lintao Zhang[3] and Paul Barham[2]

[1]University of Cambridge, Computer Laboratory, Cambridge, UK
[2]Microsoft Research, Cambridge, UK
[3]Microsoft Research, Silicon Valley, CA, USA

{Manuel.Costa,Jon.Crowcroft}@cl.cam.ac.uk

{manuelc,mcastro,antr,lidongz,lintaoz,pbar}@microsoft.com

ABSTRACT

Worm containment must be automatic because worms can spread too fast for humans to respond. Recent work has 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 approach to contain worms automatically that addresses these limitations. Vigilante relies on collaborative worm detection at end hosts, but does not require hosts to trust each other. Hosts run instrumented software to detect worms and broadcast self-certifying alerts (SCAs) upon worm detection. SCAs are proofs of vulnerability that can be inexpensively verified by any vulnerable host. When hosts receive an SCA, they generate filters that block infection by analysing the SCA-guided execution of the vulnerable software. We show that Vigilante can automatically contain fast-spreading worms that exploit unknown vulnerabilities without blocking innocuous traffic.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.4.5 [**Operating Systems**]: Reliability; D.4.8 [**Operating Systems**]: Performance; D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Security, Reliability, Performance, Algorithms, Design, Measurement

Keywords

Worm containment, Data flow analysis, Control flow analysis, Self-certifying alerts

SOSP'05, October 23–26, 2005, Brighton, United Kingdom.

1. INTRODUCTION

Worm containment must be automatic to have any chance of success because worms spread too fast for humans to respond [27, 42]; for example, the Slammer worm infected more than 90% of vulnerable hosts in 10 minutes [26].

Recent work on automatic containment [22, 38, 24, 46] has explored *network-level* approaches. These rely on heuristics to analyze network traffic and derive a packet classifier that blocks or rate-limits forwarding of worm packets. It is hard to provide guarantees on the rate of false positives and false negatives with these approaches because there is no information about the software vulnerabilities exploited by worms at the network level. False positives may cause network outages by blocking normal traffic, while false negatives allow worms to escape containment. We believe that automatic containment systems will not be deployed unless they have a negligible false positive rate.

This paper proposes a new system called Vigilante that addresses these limitations by using an *end-to-end* approach to contain worms automatically. Since hosts run the software that is vulnerable to infection by worms, they can instrument the software to gather information that is not available to network-level approaches. Vigilante leverages this information to contain worms that escape network-level detection and to eliminate false positives.

Vigilante introduces the concept of a self-certifying alert (SCA). An SCA is a machine-verifiable proof of a vulnerability: it proves the existence of a vulnerability in a service and can be inexpensively verified.

SCAs remove the need for trust between hosts. This enables cooperative worm detection with many detectors distributed all over the network, thereby making it hard for the worm to avoid detectors or to disable them with denial-of-service attacks. Additionally, cooperation allows hosts to run expensive detection engines with high accuracy [21, 4, 30, 9, 28] because it spreads detection load. For example, a host that does not run a database server can run a version of the server instrumented to detect infection attempts in a virtual machine. This instrumented version is a honeypot; it should not receive normal traffic. Therefore, the host will incur little overhead for running the detection engine, whereas a production database server would incur an unacceptable overhead.

SCAs also provide a common language to describe vulnerabilities and a common verification mechanism, which can

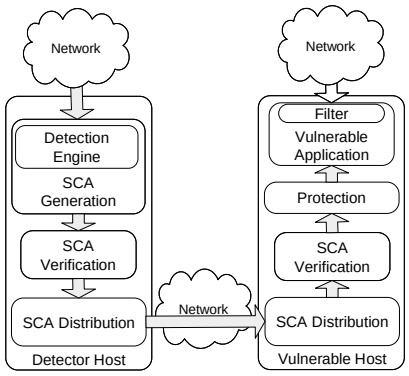


Figure 1: Automatic worm containment in Vigilante.

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, hosts detect worms by instrumenting network-facing services to analyze infection attempts. Detectors use this analysis to generate SCAs automatically and they distribute them to other hosts. 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 is certain that the service is vulnerable; the verification procedure has no false positives.

Alerted hosts protect themselves by generating filters that block worm messages before they are delivered to a vulnerable service. These filters are generated automatically using dynamic data and control flow analysis of the execution path a worm follows to exploit the vulnerability described in an SCA. Each vulnerable host runs this procedure locally and installs the filter to protect itself from the worm. These filters introduce low overhead, have no false positives, and block all worms that follow the same execution path to gain control. Figure 1 illustrates automatic worm containment in Vigilante.

We have implemented a prototype of Vigilante for x86 machines running Windows. The paper presents results of experiments using three infamous real worms: Slammer, CodeRed, and Blaster. These results show that Vigilante can effectively contain fast-spreading worms that exploit unknown vulnerabilities without blocking innocuous traffic.

The key contributions of this paper are:

- the concept of SCAs and the end-to-end automatic worm containment architecture it enables,
- mechanisms to generate, verify, and distribute SCAs automatically, and
- an automatic mechanism to generate host-based filters that block worm traffic.

Section 2 introduces the concept of SCA and describes procedures to verify, generate, and distribute SCAs. The automatic filter generation mechanism is presented in Section 3. Section 4 presents our experimental results and Section 5 describes related work. We conclude in Section 6.

2. SELF-CERTIFYING ALERTS

This section describes the format of SCAs, as well as the mechanisms to verify, generate, and distribute alerts.

2.1 Alert types

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

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

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

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

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

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

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

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

Service: Microsoft SQL Server 8.00.194
Alert type: Arbitrary Execution Control
Verification Information: Address offset 97 of message 0
Number messages: 1
Message: 0 to endpoint UDP:1434
Message data: 04,41,41,41,41,42,42,42,42,43,43,43,43,44,44,44,44,45,45,45,45,46,46,46,46,47,47,47,47,48,48,48,48,49,49,49,49,4A,4A,4A,4A,4B,4B,4B,4B,4C,4C,4C,4C,4D,4D,4D,4D,4E,4E,4E,4E,4F,4F,4F,4F,50,50,50,50,51,51,51,51,52,52,52,52,53,53,53,53,54,54,54,54,55,55,55,55,56,56,56,56,57,57,57,57,58,58,58,58,0A,10,11,61,EB,0E,41,42,43,44,45,46,01,70,AE,42,01,70,AE,42,.....

Figure 2: An example arbitrary execution control SCA for the Slammer worm. 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.

Figure 2 shows an example arbitrary execution control SCA generated for the Slammer worm. The SCA identifies the vulnerable service as Microsoft SQL Server version 8.00.194 and the alert type as an arbitrary execution control. The 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.

2.2 Alert verification

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

To verify SCAs, each host runs a virtual machine with a *verification manager* and instrumented versions of network-facing services. Each service is instrumented by loading a new library into its address space with a `Verified` function that signals verification success to the verification manager. In addition, critical functions (e.g., `exec` system calls) are wrapped using a binary rewriting tool [20]. The wrappers call `Verified` if the actual value of a critical argument matches a reference value specified by the verification manager. Otherwise, they call the original functions. Since we do not require access to the source code of the services, we can instrument any service. The host also runs an *SCA verifier* process outside the virtual machine that provides other processes with an interface to the verification module and acts as a reverse firewall to ensure containment.

Figure 3 illustrates the SCA verification procedure. When the SCA verifier receives an SCA for verification, it sends the SCA to the verification manager inside the virtual machine. The verification manager uses the data in the SCA to identify the vulnerable service. Then it modifies the sequence of messages in the SCA to trigger execution of `Verified` when the messages are sent to the vulnerable service. The modifications involve changing the byte string at the offset of the message specified in the verification information according to the alert type. This byte string is changed to:

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

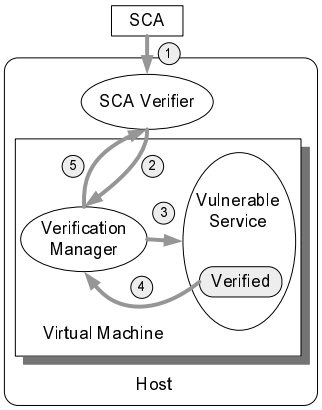


Figure 3: SCA verification.

- the code for `call Verified` for arbitrary code execution alerts,
- or the reference critical argument value for arbitrary function argument alerts.

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

The state of the virtual machine is saved to disk before any verification is performed. This reference state is used to start uncompromised copies of the virtual machine for verification. After performing a verification, the virtual machine is destroyed and a new one is started from the reference state in the background to ensure that there is a virtual machine ready to verify the next SCA. The experimental results in Section 4 show that the memory and CPU overheads to keep the virtual machine running are small.

Vigilante's alert verification procedure has three important properties:

Verification is fast. The time to verify an SCA is similar to the time it takes the worm to infect the service because the overhead of the instrumentation and the virtual machine are small. Additionally, we keep a virtual machine ready to verify SCAs. This is important for ensuring timely distribution of alerts.

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

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

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

tending our implementation to specify a conversion function for these values in SCAs.

Second, the current implementation assumes that sending the sequence of messages in an SCA to the vulnerable service is sufficient to replay the exploit during verification. This is true for all previous worms that we are aware of, but it may be insufficient for some worms. For example, the success of some exploits may depend on a particular choice of scheduling order for the threads in a service. We could address this limitation by including other events in SCAs (e.g., scheduling events and other I/O events) and by replaying them during verification. There is a large body of work in this area [14, 13] that we could leverage.

2.3 Alert generation

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

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

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

There are many engines to detect worms at the host level with different tradeoffs between coverage and overhead [47, 10, 23, 21, 16, 4, 9, 30, 2]. We implemented SCA generation for two different detection engines: non-executable (*NX*) pages [1] and dynamic dataflow analysis [43, 9, 11, 30]. We chose these engines because they represent extreme points in the tradeoff between coverage and overhead: the first detector has low overhead but low coverage whereas the second has high overhead and high coverage. Furthermore, neither of them require access to source code, which makes them widely applicable.

2.3.1 Non-executable pages

The first detection engine uses non-execute protection on stack and heap pages to detect and prevent code injection attacks. It has negligible runtime overhead with emerging hardware support and has relatively low overhead even when emulated in software [1]. This detector can be used to generate arbitrary execution control or arbitrary code execution SCAs as follows.

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

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

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

2.3.2 Dynamic dataflow analysis

Dynamic dataflow analysis is a generic detection engine that has been proposed concurrently by us [9] and others [43, 11, 30]. It can be used to generate the three types of alerts discussed in the previous sections.

The idea is to track the flow of data received in certain input operations; for example, data received from network connections. This data and any data derived from it is marked dirty. The engine blocks dangerous uses of dirty data and signals attempts to exploit vulnerabilities:

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

Vigilante implements dynamic dataflow analysis on x86 CPUs using binary re-writing [25] at load time. We instrument every control transfer instruction (e.g., RET, CALL, JMP), every critical function, and every data movement instruction (e.g., MOV, MOVS, PUSH, POP).

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 dynamic dataflow algorithm is simple: whenever an instruction that moves data from a source to a destination

```
mov al,byte ptr [msg]          //move first byte to AL
add al,0x10                    //add 0x10 to AL
mov cl,0x31                    //move 0x31 into CL
cmp al,cl                      //compare AL to CL
jne out                        //jump if not equal
mov cl,byte ptr [msg]          //move first byte to CL
xor eax,eax                    //move 0x0 into EAX
loop:
mov byte ptr [esp+eax+4],cl    //move byte into buffer
mov cl,byte ptr [eax+msg+1]    //move next byte to CL
inc eax                        //increment EAX
test cl,cl                     //test if CL equals 0x0
jne loop                       //jump if not equal
out:
mov esp,ebp
ret
```

Figure 4: Vulnerable code.

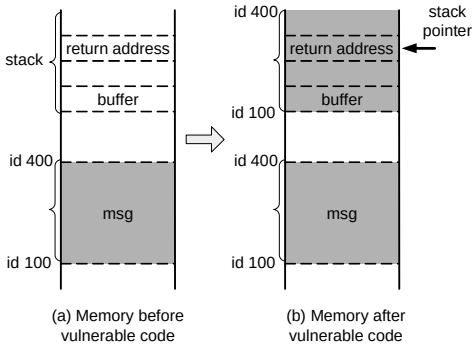


Figure 5: Example of SCA generation with dynamic dataflow analysis. The figure shows the memory when (a) a message is received and the vulnerable code is about to execute, and (b) after the vulnerable code executes and overwrites the return address in the stack. Grayed areas indicate dirty memory regions and the identifiers of dirty data are shown on the left.

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

We will use the vulnerable code in Figure 4 as an example to illustrate SCA generation with dynamic dataflow analysis. We assume that the buffer `msg` contains a message received from the network. The code starts by adding `0x10` to the first byte in the message and then comparing the result with a constant (`0x31`). If they match, the bytes in `msg` are copied to a stack-based buffer until a zero byte is found. This is a potential buffer overflow that could overwrite the return address on the stack and it is representative of vulnerabilities in string libraries.

Figure 5 shows the state of memory before and after the vulnerable code is executed. In this example, the bytes in the incoming attack message were mapped to identifiers from 100 to 400. Before the code is executed, the memory region where the message was received is marked dirty with identifiers from 100 to 400. When the `ret` instruction is about to execute, a portion of the stack has also been marked dirty with identifiers from 100 to 400 because the message data was copied to the stack buffer by the instrumented data movement instructions. Since the copy overwrites 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. The message in the verification information is the attack message because the identifier of the dirty data falls in the 100 to 400 range, 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.

Dynamic data flow analysis suffers from a small but non-negligible false positive rate, for example, an integer value in a message can be used to index a jump table after checking if it is within bounds, or a critical function can be called with a dirty critical argument that was checked by the program. It also suffers from poor performance when implemented in software [30]. Hardware implementations of dynamic dataflow analysis [43, 11] perform well but they have not been implemented yet and they lack flexibility (for example, they cannot track the origin of dirty data to aid SCA generation).

Vigilante addresses both of these issues. Verification eliminates false positives and the cooperative detection architecture spreads the detection load.

2.4 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 [6] to broadcast SCAs.

Vigilante uses flooding to broadcast SCAs to all the hosts in the overlay: each host sends the SCA to all its overlay 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 [12] by requiring each host to have a certificate signed by a trusted offline certification authority to participate in the overlay [6]. 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 [6]. 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 they are not able to increase the number of overlay paths through compromised hosts.

Compromised hosts in the overlay may also attempt to disrupt SCA distribution with denial of service attacks. Vigilante uses three techniques to mitigate these attacks: hosts do not forward SCAs that are blocked by their filters or are identical to SCAs received recently, they only forward SCAs that they can verify, and they impose a rate limit on the number of SCAs that they are willing to verify from each neighbor. The first technique prevents attacks that flood variants of old SCAs and the second prevents attacks that flood bogus SCAs to all the hosts in the overlay. Since hosts only accept SCAs received over the authenticated connections to their 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 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 that there is a large number of disjoint overlay paths between each pair of nodes. Since flooding explores all paths in the overlay, the probability that SCAs are delivered to vulnerable nodes is very high even when the fraction of nodes that can verify the SCA is small.

Additionally, verifying SCAs introduces delay. Our verification procedures are fast but the attacker can increase delay with denial of service attacks. In addition to the techniques above, we verify SCAs from different 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 information about the overlay topology from the worm. One technique to achieve this is to run the overlay code in a separate virtual machine and to enforce a narrow interface that does not leak information about the addresses of overlay neighbors.

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

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

A secure overlay with super-peers is the best option for SCA distribution and we believe it is deployable. It could be supported easily by an infrastructure similar to Akamai's, which is already used by anti-virus companies to distribute signatures [3].

3. LOCAL COUNTERMEASURES

Hosts can take local actions to protect themselves when they receive an SCA. For example, they can stop the vulnerable service or run it with a detection engine to prevent infection. Stopping the program is not acceptable in most settings and running a high-coverage detection engine (e.g., dynamic dataflow 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.

Vigilante uses host-based filters to block worm traffic before it is delivered to the vulnerable service. These filters are unlikely to affect the correct behavior of the service because they do not change the vulnerable service and they allow the service to continue running under attack. Host-based filters have been proposed before, for example, in Shield [45]. The novelty is that we describe a mechanism to generate these filters automatically such that they have no false positives, are effective at blocking worm traffic, and introduce very low overhead.

3.1 Automatic filter generation

Before the host attempts to generate a filter, it verifies

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. Therefore, the host generates a filter for the exploit described in the SCA and suspends the vulnerable program to prevent infection during the filter generation process. If the verification fails, the SCA is dropped and the host does not consume resources generating a filter. This is important for mitigating denial-of-service attacks because verification is significantly cheaper than filter generation.

Hosts generate filters automatically by analyzing the execution path followed when the messages in the SCA are replayed. They use 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 x86 instructions 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, and also conditions used when executing conditional move and set instructions. We call the conjunction of these conditions the *filter* condition. The filter condition is initially *true* and it is updated after every instruction that uses a dirty processor flag or transfers control to an address read from a dirty location. The filter condition is updated to be the conjunction of its old value and the appropriate conditions on the expressions computed by the data flow graphs of the dirty flag and address location.

For example, when the instruction `jz addr` 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` is executed, the filter condition is left unchanged if the `eax` register is clean. If `eax` is dirty, we add the condition that the expression computed by `eax`'s data flow graph be equal to the value currently stored by `eax`.

We will use the vulnerable code in Figure 4 and the corresponding arbitrary execution control SCA from Section 2.3 to illustrate the filter generation procedure. 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*. After executing the first instruction `al` would be dirty and its data flow graph would be `msg[0]`. After the second instruction, `al` would remain dirty and its data flow graph would change to `msg[0] + 0x10`. The zero flag would become dirty after the fourth instruction and its data flow graph would become `msg[0] + 0x10=0x31`. Therefore, the filter condition would be updated to `msg[0] + 0x10=0x31`

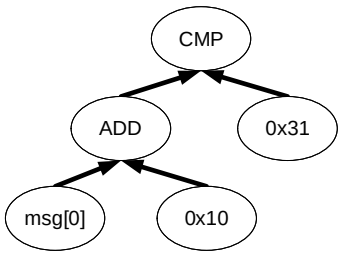


Figure 6: Dataflow graph for the zero flag when the instruction `jne out` is executed by the program in Figure 4.

after the fifth instruction because the jump is not taken. Similarly, executing each iteration of the loop would add a condition of the form `msg[i]`$\neq$`0` for `i` $>$ `0`.

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

In the current implementation, each data flow graph has constants, byte offsets in messages, and x86 opcodes as vertices and the edges connect the operands of an instruction with its opcode. For example, Figure 6 shows the dataflow graph associated with the zero flag when the `jne out` instruction is executed (in the example in Figure 4). The filter condition is represented as a list of graphs with the same format. Therefore, the filter condition can be translated into efficient executable x86 code for filtering incoming messages. Figure 7 shows the translation of the dataflow graph in Figure 6 into x86 assembly code. The translation is carried out by doing a depth-first traversal of the graph to generate a stack-based evaluation of the dataflow expression.

The code generated for the filters is safe. We ensure that it 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 dataflow expressions. Filters also check that a message is at least as long as the largest offset used by the filter code.

```
mov esi, msg                  //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
add al,bl                   //add 0x10 to al
push eax
push 0x31
pop ebx
pop eax
cmp eax, ebx                  //compare with 0x31
jne do_not_drop               //if not equal, do not drop msg
```

Figure 7: Filter code generated automatically for the data flow graph in Figure 6.

Furthermore, the filter code has no loops since it includes only forward jumps.

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. Additionally, they can filter many worm variants that exploit the same vulnerability because the filter captures the exact conditions that determine the path to exploit the vulnerability. These filters are very different from filters that block messages that contain a particular string [22, 38] or sequence of strings [29]. They can capture arbitrary computations on the values of the input messages.

3.2 Two filters to reduce false negatives

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 non-zero bytes. This filter would not block variants of the worm that used smaller messages.

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

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

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

The first heuristic removes conditions on message bytes that appear after the offset identified by the verification information in the SCA. Since the bytes in the message are usually processed in order, this heuristic is unlikely to introduce false positives. The second heuristic removes conditions added by the execution of a function when that function returns. The rationale is that these conditions are usually not important after the function returns and that the important effects of the function are captured in the data flow graphs of dirty data. 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.

We are working on combining static analysis techniques (e.g., program chopping [33]) with our dynamic analysis to generate specific filters that are more general but are guaranteed not to have false positives. We are also studying other heuristics to create general filters that can capture even more worm variants but still have low false positive rate. For example, we believe that we can generate token sequence filters similar to Polygraph [29] from a single worm variant.

4. EVALUATION

We implemented a prototype of Vigilante for x86 machines running Windows. This section evaluates the performance of our prototype and describes implementation details.

4.1 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. These worms attacked popular services and had a high impact on the Internet.

Slammer infected approximately 75,000 Microsoft SQL Servers. It was the fastest computer worm in history [26]. During its outbreak, the number of infected machines doubled every 8.5 seconds. Slammer's exploit uses a UDP packet with the first byte set to 0x04 followed by a 375 byte string with the worm code. While copying the string, SQL overwrites a return address in the stack.

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

Blaster infected the RPC service on Microsoft Windows machines. We conservatively estimate that it infected 500,000 hosts and that its spread rate was similar to CodeRed's. Blaster is a two-message attack: the first message is an

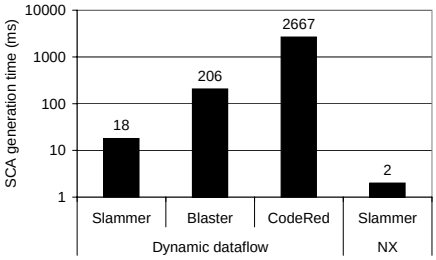


Figure 8: SCA generation time in milliseconds for real worms using two detectors.

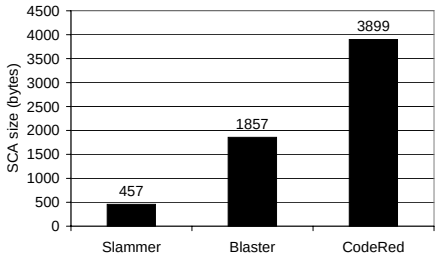


Figure 9: SCA sizes in bytes for real worms.

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.

Experiments with CodeRed and Blaster ran on Windows 2000 Server and experiments with Slammer ran on Windows XP with SQL Server 2000.

4.2 Alert generation

SCA generation with dynamic dataflow analysis relies on binary instrumentation at load time using Nirvana [25]. Nirvana dynamically modifies code and injects instrumentation at run time. It operates on normal binaries and does not require availability of symbol information. Nirvana translates code sequences dynamically into instrumented versions that are kept in a code cache in a manner similar to DynamoRIO [5]. This instrumentation ensures that the detection engine is invoked before every instruction to disassemble the instruction and examine its operands. The engine updates the data structures that keep track of dirty data and its origin when data movement instructions are executed. 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 2.3). Vigilante intercepts socket operations to log received data and to mark the socket buffers dirty.

The first experiment measures the time to generate SCAs with the detectors in Section 2.3. The time is measured from the moment the last worm message is received till the detector generates an SCA. It does not include the time to verify the SCA before it is distributed and the log contains only the worm messages. One detector uses dynamic dataflow analysis and the other uses a software emulation of non-execute protection on stack and heap pages (*NX*). The detectors generate arbitrary execution control alerts for Slammer and Blaster, and an arbitrary code execution alert for CodeRed.

Figure 8 shows average SCA generation times for Slammer, Blaster, and CodeRed with the dynamic dataflow detector and for Slammer using the NX detector. The results are the average of five runs. The standard deviation is 0.5 ms for Slammer, 3.9 ms for Blaster, and 204.7 ms for CodeRed.

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

Figure 9 shows the SCA size in bytes for each worm. The SCAs include a fixed header of 81 bytes that encodes the SCA type, vulnerable service identification and verification information. The size of the SCAs is small and it is mostly determined by the size of the worm probe messages.

4.3 Alert verification

SCAs are verified inside a Virtual PC 2004 virtual machine (VM) to isolate any side-effects of the verification process (see Figure 3). The SCA verifier communicates with the verification manager through a virtual network connection. During the initial VM setup phase, a dynamic link library (DLL) with the `Verified` function is loaded into network-facing services and the initialization routine for the library reports the address of `Verified` to the verification manager through a shared memory section. 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.

When an SCA arrives, the verification manager replays the messages in the SCA (as described in Section 2.2) and waits on a synchronization object. If the SCA is valid, the `Verified` function executes and sets the synchronization object. This signals success to the verification manager who sends a success notification message to the SCA verifier.

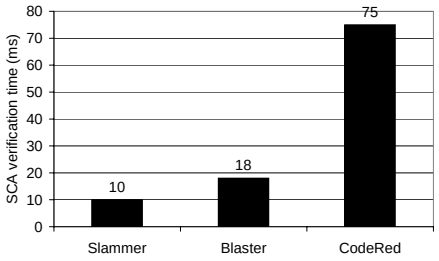


Figure 10: SCA verification time in milliseconds for real worms.

Figure 10 shows the average time in milliseconds to verify each SCA. The results are the average of five runs. The standard deviation is 0.5 ms for Slammer, 1.5 ms for Blaster, and 6.5 ms for CodeRed.

Verification is fast because we keep a VM running that is ready to verify SCAs when they arrive. The VM has all the code needed for verification loaded. The overhead to keep the VM running is low: a VM with all vulnerable services used less than 1% of the CPU and consumed approximately 84MB of memory.

We also explored the possibility of starting VMs on demand to verify SCAs. The VM is compressed by the Virtual PC into a 28MB checkpoint. It takes four seconds to start the VM from disk with cold caches, but it takes less than a second to start the VM from a RAM disk. Since this additional delay is problematic when dealing with fast spreading worms, we decided to keep a VM running. We are investigating techniques to fork running services [15] that should enable creation of VMs on demand with low delay.

4.4 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.

4.4.1 Simulation setup

The simulations ran on a simple packet-level discrete event simulator with a transit-stub topology generated using the Georgia Tech topology generator [49]. 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 core routers is computed by the topology generator and routing is performed using the routing policy weights of the graph generator. Vigilante hosts are attached to randomly selected stub routers by a LAN link with a delay of 1 ms.

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

We model worm propagation using the epidemic model described in [19] 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:

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

$$\frac{dI_t}{dt} = \beta\ I_t(1 - p - \frac{I_t}{S}) \quad (2)$$

Starting with k initially infected hosts, whenever a new vulnerable host is infected at time t, our simulator calculates the expected time a new susceptible host receives a worm probe based on Equations (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 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 [17]. At the time, the TTM service had measurement hosts at 54 sites spread across the world and each host sent a probe to each of the other hosts every 30 seconds.

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

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

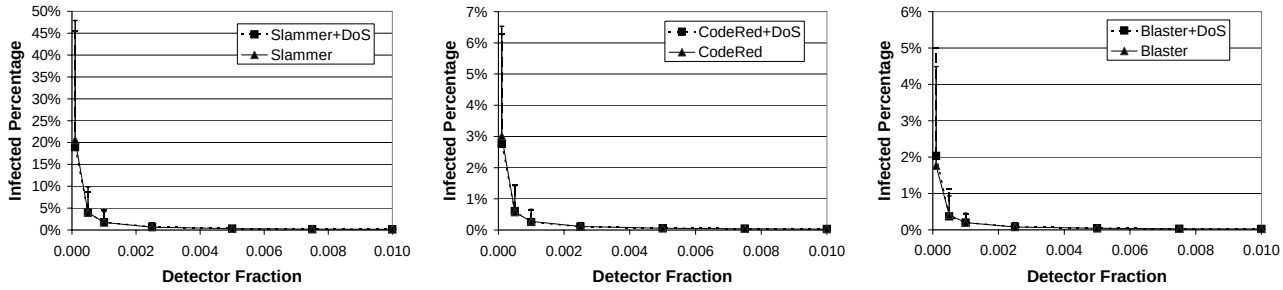
	β	S	T_g (ms)	T_v (ms)
Slammer	0.117	75,000	18	10
CodeRed	0.00045	360,000	2667	75
Blaster	0.00045	500,000	206	18

Table 1: Simulation parameters for modeling containment of real worms.

4.4.2 Containment of real worms and beyond

First, we evaluate the effectiveness of Vigilante with Slammer, CodeRed, and Blaster. Table 1 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. [26]. 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 4.2 and 4.3.

Figure 11 shows the infected percentage (i.e., the percentage of vulnerable hosts that are eventually infected by the worm) for the real worms with different fractions (p) of detectors both with and without DoS attacks. The graph shows that a small fraction of detectors ($p = 0.001$) is enough to contain the worm infection to less than 5% of the vulnerable population, even under DoS attacks. The Vigilante overlay is extremely effective in disseminating SCAs: once a detector is probed, it takes approximately 2.5 seconds (about 5 overlay hops) to reach almost all the vulnerable hosts.



(a) Slammer (b) CodeRed (c) Blaster

Figure 11: Containment of Slammer, CodeRed, and Blaster using parameter settings in Table 1, both with and without DoS attacks. Each data point is the mean value with an error bar up to the 90^{th} percentile value.

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

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

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

4.5 Filters

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

4.5.1 Filter generation

The first experiment measures the time to generate a filter from an SCA that has already been verified. Figure 13 shows the time in milliseconds to generate both the specific and general filters for the three worms. The results are the average of five runs. The standard deviation was 0.7 ms for Slammer, 5.1 ms for Blaster, and 205.3 ms for CodeRed. In all cases, filter generation is fast. Filter generation for CodeRed is more expensive because the number of instructions analyzed is larger and the binary re-writing tool needs to dynamically translate code for a number of libraries that are loaded on demand.

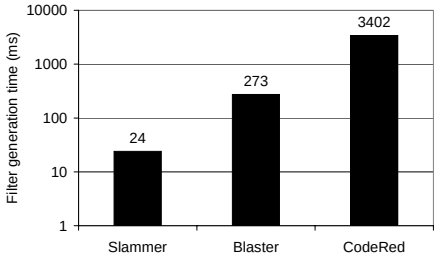


Figure 13: Filter generation time for real worms.

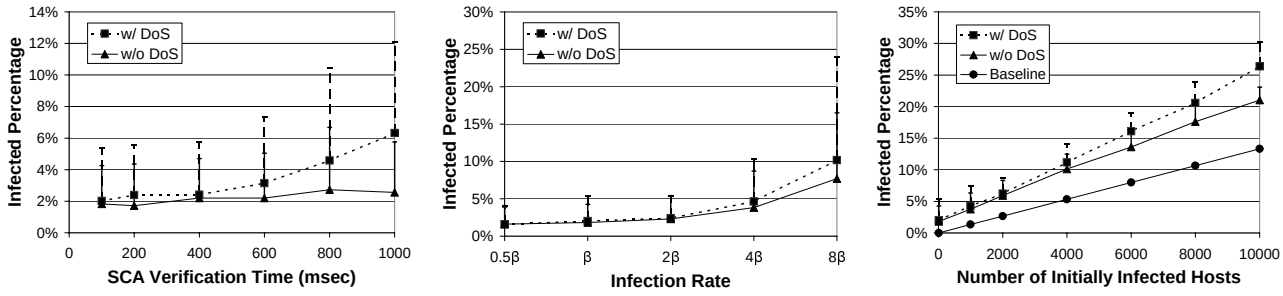
```
xor eax,eax                  //clear EAX register
mov al,byte ptr [esi + 0x0]  //move first byte into AL
push eax          //push the first byte into the stack
push 0x02
pop ebx
pop eax
sub eax,ebx        //subtract 2 from first byte
push eax
pop eax
mov ebx,0x02
cmp eax,ebx        //compare with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

Figure 14: x86 code for Slammer filter's first condition.

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

The general filter for Slammer checks that the first byte is 0x4 and that the following bytes are non-zero (up to the byte offset of the value that would overwrite the return address in the stack). This filter is optimal: it captures all polymorphic variations of the attack with no false positives. The filter's code sequence is not optimized: it corresponds to a stack-based evaluation of the filter condition. For example in Slammer, the condition that the first byte is equal to 0x04 is computed by the code in Figure 14. 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 '\'



(a) (b) (c)

Figure 12: The effect of SCA verification time, infection rate, and number of initially infected hosts, both with and without DoS attacks. Each data point is the mean value with an error bar up to the 90^{th} percentile value.

up to the position of the value that will overwrite the return address in the stack. This filter catches all polymorphic variations in the worm code and some variations in other parts of the message.

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

These results show that dynamic control and data flow analysis is a promising approach to filter generation. While the general Slammer filter is perfect, the general Blaster and CodeRed filters 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 [32].

4.5.2 Overhead of deployed filters

We also measured the performance overhead introduced by deployed filters. Filters were deployed by binary rewriting the vulnerable services. We used Detours [20] to intercept calls to the socket interface and install the filters immediately above functions that receive data.

We ran three experiments for each vulnerable service and measured the overhead with a sampling profiler. The first experiment (*intercepted*) ran the service with just the socket interface being intercepted. The second experiment (*intercepted + filter*) ran the service with the socket interface being intercepted and invoking the appropriate general and specific filters. The third experiment (*intercepted + filter + attack*) stressed the filter code by sending worm probes to the service at a rate of 10 per second (which is three orders of magnitude larger than the rate induced by Slammer). For every experiment, we increased the service load

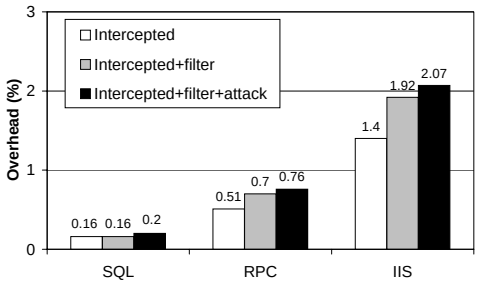


Figure 15: CPU overhead of network traffic interception and filter deployment.

until it reached 100% CPU usage, as described below. Figure 15 shows the overhead for each of the experiments for the three vulnerable services. The results are the average of five runs. The overhead is very low in all cases.

SQL For Slammer the vulnerable service is SQL Server. We generated load using the TPC-C benchmark [44] 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 15 shows that the CPU consumed by the interception is just 0.16%. When then Slammer filters are installed, the overhead remains the same because Slammer exploits a vulnerability in a management interface running on UDP port 1434. This is not the same communication endpoint that SQL uses to listen for client transactions. Therefore, the requests issued by the TPC-C clients follow a different code path and the impact of running the filter is negligible. With worm probes, the overhead rises to only 0.2%.

RPC For Blaster the vulnerable service is Microsoft Windows RPC service. We generated a custom workload using requests to lookup and register an RPC interface. We loaded the RPC service using 3 client hosts that lookup the RPC interface and 1 local client that registers the interface. Figure 15 shows the CPU consumed by interception is only 0.51%, and it rises to 0.7% when the filters are invoked. When running with 10 Blaster probes per second the overhead was 0.76%. Unlike Slammer, the filters are on the

normal execution path and are used by requests to lookup the interface.

IIS For CodeRed the vulnerable service is Microsoft IIS Server. We generated a workload using the requests from the SpecWeb99 [40] 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 15 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%.

4.6 End-to-End experiments

The final set of experiments measures Vigilante's worm containment process end-to-end in a five-host Vigilante network. The hosts were configured in a chain representing a path from a detector to a vulnerable host in the SCA distribution overlay with three super-peers. They were connected by a LAN. The first host was a detector running a dynamic dataflow analysis engine. Once the detector generated an SCA it was propagated through three super-peers to a host running the vulnerable service. This provides approximately the same number of hops as the simulations in Section 4.4.

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

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

5. RELATED WORK

There has been much work on worm containment systems. Much of it has been based on generating content signatures or detecting abnormal communication patterns.

Worm signatures have traditionally been generated by humans but there are several recent proposals to generate signatures automatically [24, 22, 38]. These systems can generate a signature for an unknown worm by identifying a common byte string in suspicious network flows. Newsome et al. [29] study the ability of these systems to contain polymorphic worms. They conclude that a single byte string is not enough but signatures with a set of strings can contain some polymorphic worms. However, the absence of information about software vulnerabilities at the network level makes it hard to provide guarantees on the rate of false positives.

Another approach to contain worms automatically is based on blocking or rate limiting traffic from hosts that exhibit abnormal communication patterns. Williamson [48] proposed limiting the rate of connections to new destinations. Snort [35] and Network Security Monitor [18] detect worm traffic by monitoring the rate at which unique destination addresses are contacted and they block the sender. Bro [31] uses a configurable threshold on the number of failed connections and Weaver [46] uses a threshold on the ratio of failed to successful connections. Traffic from hosts that exceed these thresholds is blocked. These systems cannot contain worms that have normal traffic patterns, for example, topological worms that exploit information about hosts in infected machines to propagate, or slow-spreading worms that do not generate connections at abnormal rates. They can have false positives, for example, an attacker can perform scanning with a fake source address to block traffic from that address.

The work in [36] has proposed a host-based architecture to contain worms automatically. Their architecture is missing the key concept of SCAs. Each organization runs a central service that generates patches automatically using a set of heuristics to modify vulnerable source code, for example, modifying the code to move vulnerable buffers to the heap. We believe that Vigilante's architecture is more resilient to attack, because hosts can protect themselves automatically by generating filters that are more general than the heuristics proposed in [36] and are less likely to affect the correct running of the vulnerable services.

Several systems have proposed mechanisms that, like Vigilante filters, allow vulnerable services to continue execution while being attacked. Rinard et al. [34] propose using a C compiler that inserts runtime checks for illegal memory accesses. Their system discards invalid memory writes and manufactures values for invalid reads. Sidiroglou et al. [37] propose 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. DIRA [39] is a GCC extension that checks for overwrites of control data structures and allows rolling back vulnerable services to the entry point of a function. Vigilante filters are more efficient than these techniques and they are less likely to affect the correct execution of the protected services.

Several systems provide interesting alternatives to deploy Vigilante filters. IntroVirt [7] uses vulnerability-specific predicates to analyze the execution state of applications and operating systems running inside virtual machines. Like Vigilante filters, IntroVirt predicates can compute generic conditions, but they are generated manually for known vulnerabilities. By using virtual machine rollback and replay, IntroVirt is able to detect if vulnerabilities were exploited in the past. We could deploy Vigilante filters as IntroVirt predicates. Shield [45] uses host-based filters to block vulnerabilities but these filters are generated manually. We could use Shield's infrastructure to deploy our filters.

Several authors have proposed models for predicting worm propagation speeds and for analyzing defense mechanisms [42,

27, 50, 8, 41]. Flash worms are the fastest theoretically predicted worms. As these worms become better understood, we plan to study their containment.

There is a large amount of work on host-based detection of vulnerabilities including [47, 10, 23, 21, 16, 2, 4]. We could use these as detectors to generate SCAs in Vigilante.

6. CONCLUSIONS

Worm containment must be automatic but 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 adopts an end-to-end approach 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 the fundamental 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 and provide a common language to describe vulnerabilities and a common mechanism to verify alerts.

Vigilante also introduces a new mechanism to generate host-based filters automatically by performing dynamic data and control flow analysis of attempts to infect programs. These filters can block worms with no false positives and they are effective at containing worms that exploit a large class of vulnerabilities.

Our experimental results show that Vigilante can contain real worms like Slammer, Blaster, CodeRed, and polymorphic variants of these worms. They also show that Vigilante can contain worms that propagate faster than Slammer even when only a small fraction of hosts can detect the worm.

7. ACKNOWLEDGEMENTS

We thank our shepherd, David Culler, the anonymous reviewers, and Steve Hand for their comments. We thank Sanjay Bhansali and Darek Mihocka for their help with Nirvana. We thank Eric Traut for discussions about Virtual PC. We thank Rene Wilhelm and Henk Uijterwaal for access to the RIPE data. We thank Jacob Gorm Hansen for implementing the Windows NX detector.

8. REFERENCES

[1] Pax team. `http://pax.grsecurity.net/`.

[2] Abadi, M., Budiu, M., Erlingsson, U., and Ligatti, J. Control-flow Integrity: Principles, implementations, and applications. In *ACM CCS* (Nov. 2005). *To appear.*

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

[4] Barrantes, E. G., Ackley, D. H., Palmer, T. S., Stefanovic, D., and Zov, D. D. Randomized instruction set emulation to disrupt binary code injection attacks. In *ACM CCS* (Oct. 2003).

[5] Bruening, D., Duesterwald, E., and Amarasinghe, S. Design and implementation of a dynamic optimization framework for Windows. In *ACM FDD0* (Dec. 2000).

[6] Castro, M., Druschel, P., Ganesh, A., Rowstron, A., and Wallach, D. S. Security for structured peer-to-peer overlay networks. In *OSDI* (Dec. 2002).

[7] Chen, P., Joshi, A., King, S., and Dunlap, G. Detecting past and present intrusions through vulnerability-specific predicates. In *SOSP* (Oct. 2005).

[8] Chen, Z., Gao, L., and Kwiat, K. Modelling the spread of active worms. In *IEEE INFOCOM* (Apr. 2003).

[9] Costa, M., Crowcroft, J., Castro, M., and Rowstron, A. Can we contain Internet worms? In *HotNets* (Nov. 2004).

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

[11] Crandall, J. R., and Chong, F. T. Minos: Control data attack prevention orthogonal to memory model. In *MICRO-37* (Dec. 2004).

[12] Douceur, J. R. The Sybil attack. In *IPTPS* (Mar. 2002).

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

[14] Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys 34*, 3 (Sept. 2002), 375–408.

[15] Fraser, K., and Chang, F. Operating System I/O Speculation: How two invocations are faster than one. In *USENIX Annual Technical Conference* (Jun. 2003).

[16] Garfinkel, T., and Rosenblum, M. A virtual machine introspection based architecture for intrusion detection. In *NDSS* (Feb. 2003).

[17] Georgatos, F., Gruber, F., Karrenberg, D., Santcroos, M., Uijterwaal, H., and Wilhelm, R. Providing Active Measurements as a Regular Service for ISPs. In *PAM2001* (Apr. 2001). `http://www.ripe.net/ttm`.

[18] Heberlein, L. T., Dias, G., K, L., Wood, B. M. J., and Wolber, D. A network security monitor. In *Proceedings of the IEEE Symposium on Research in Privacy* (1990).

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

[20] Hunt, G., and Brubacher, D. Detours: Binary interception of Win32 functions. In *USENIX Windows NT Symposium* (July 1999).

[21] Kc, G. S., Keromytis, A. D., and Prevelakis, V. Countering code-injection attacks with instruction-set randomization. In *ACM CCS* (Oct. 2003).

[22] Kim, H., and Karp, B. Autograph: Toward automated, distributed worm signature detection. In *USENIX Security Symposium* (Aug. 2004).

[23] Kiriansky, V., Bruening, D., and Amarasinghe, S. P. Secure execution via program shepherding. In *USENIX Security Symposium* (Aug. 2002).

[24] Kreibich, C., and Crowcroft, J. Honeycomb - creating intrusion detection signatures using honeypots. In *HotNets* (Nov. 2003).

[25] Microsoft. Nirvana. `http://www.microsoft.com/windows/cse/bit.mspx`.

[26] Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., and Weaver, N. Inside the Slammer worm. *IEEE Security and Privacy 1*, 4 (Jul. 2003).

[27] Moore, D., Shannon, C., Voelker, G., and Savage, S. Internet quarantine: Requirements for containing self-propagating code. In *IEEE INFOCOM* (Apr. 2003).

[28] Necula, G. C., McPeak, S., and Weimer, W. CCured: type-safe retrofitting of legacy code. In *POPL* (Jan. 2002).

[29] Newsome, J., Karp, B., and Song, D. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy* (May 2005).

[30] Newsome, J., and Song, D. Dynamic taint analysis: Automatic detection and generation of software exploit attacks. In *NDSS* (Feb. 2005).

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

[32] Ptacek, T. H., and Newsham, T. N. Insertion, evasion, and denial of service: Eluding network intrusion detection. Tech. rep., Secure Networks, Inc, Jan. 1998.

[33] Reps, T., and Rosay, G. Precise interprocedural chopping. In *ACM SIGSOFT Symposium on Foundations of Software Engineering* (1995).

[34] Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., and Jr., W. S. B. Enhancing server availability and security through failure-oblivious computing. In *OSDI* (Dec. 2004).

[35] Roesch, M. Snort: Lightweight intrusion detection for networks. In *Conference on Systems Administration* (Nov. 1999).

[36] Sidiroglou, S., and Keromytis, A. D. Countering network worms through automatic patch generation. *IEEE Security and Privacy* (2005).

[37] Sidiroglou, S., Locasto, M. E., Boyd, S. W., and Keromytis, A. D. Building a reactive immune system for software services. In *Usenix Annual Technical Conference* (Apr. 2005).

[38] Singh, S., Estan, C., Varghese, G., and Savage, S. Automated worm fingerprinting. In *OSDI* (Dec. 2004).

[39] Smirnov, A., and cker Chiueh, T. DIRA: Automatic detection, identification, and repair of control-hijacking attacks. In *NDSS* (Feb. 2005).

[40] SPEC. Specweb99 benchmark. `http://www.spec.org/osg/web99`.

[41] Staniford, S., Moore, D., Paxson, V., and Weaver, N. The top speed of flash worms. In *WORM* (Oct. 2004).

[42] Staniford, S., Paxson, V., and Weaver, N. How to 0wn the internet in your spare time. In *USENIX Security Symposium* (Aug. 2002).

[43] Suh, G. E., Lee, J., and Devadas, S. Secure program execution via dynamic information flow tracking. In *ASPLOS XI* (Oct. 2004).

[44] TPC. Tpc-c online transaction processing benchmark. `http://www.tpc.org/tpcc/default.asp`.

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

[46] Weaver, N., Staniford, S., and Paxson, V. Very fast containment of scanning worms. In *USENIX Security Symposium* (Aug. 2004).

[47] Wilander, J., and Kamkar, M. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS* (Feb. 2003).

[48] Williamnson, M. M. Throttling viruses: Restricting propagation to defeat mobile malicious code. *ACSAC* (2002).

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

[50] Zou, C. C., Gao, L., Gong, W., and Towsley, D. Monitoring and early warning for internet worms. In *ACM CCS* (Oct. 2003).