Can we contain Internet worms?

Manuel Costa[1,2], Jon Crowcroft[1], Miguel Castro[2] and Antony Rowstron[2]

[1]*University of Cambridge, Computer Laboratory, 15 JJ Thomson Avenue, Cambridge, CB3 0FD, UK*
{Manuel.Costa,Jon.Crowcroft}@cl.cam.ac.uk
[2]*Microsoft Research Ltd., 7 J J Thomson Avenue, Cambridge, CB3 0FB, UK*
{manuelc,mcastro,antr}@microsoft.com

ABSTRACT

Worm containment must be automatic because worms can spread too fast for humans to respond. Recent work has proposed a network centric approach to automate worm containment: network traffic is analyzed to derive a packet classifier that blocks (or rate-limits) worm propagation. This approach has fundamental limitations because the analysis has no information about the application vulnerabilities exploited by worms. This paper proposes Vigilante, a new host centric approach for automatic worm containment that addresses these limitations. Vigilante relies on collaborative worm detection at end hosts in the Internet but does not require hosts to trust each other. Hosts detect worms by analysing attempts to infect applications and broadcast self-certifying alerts (SCAs) when they detect a worm. SCAs are automatically generated machine-verifiable proofs of vulnerability; they can be independently and inexpensively verified by any host. Hosts can use SCAs to generate filters or patches that prevent infection. We present preliminary results showing that Vigilante can effectively contain fast spreading worms that exploit unknown vulnerabilities.

1. INTRODUCTION

Worms can spread too fast for humans to respond [16, 22], for example, the Slammer worm infected more than 90% of vulnerable hosts in 10 minutes [15]. Recent work [11, 21, 13, 29] has proposed a *network centric* approach to automate worm containment that relies on heuristics to analyze network traffic and to derive a packet classifier that blocks (or rate-limits) forwarding of worm packets. For example, EarlyBird [21] assumes that identical byte strings in many packets from different flows are indicative of worm activity and block packets containing those strings.

The network centric approach has fundamental limitations because it is limited to heuristics that attempt to distinguish normal traffic from worm traffic. Since there is no information about the software vulnerabilities exploited by worms at the network level, these heuristics are prone to both false positives and false negatives. False positives can lead to network outages and false negatives allow worms to escape containment. For example, the heuristics in [11, 21, 13] can block normal traffic when attackers send suspect packets with strings that are common in normal traffic, and they cannot block polymorphic worms[1], which mutate their code each time they infect a new machine.

This paper proposes Vigilante, a new *host centric* approach for automatic worm containment that addresses the limitations of the network centric approach. Vigilante relies on collaborative worm detection at end hosts in the Internet but does not require hosts to trust each other. The hosts analyse attempts to infect applications and gather detailed information about the vulnerabilities exploited by worms to eliminate false positives. Hosts use this information to automatically generate self-certifying alerts (SCAs) that they broadcast to other hosts. SCAs are machine-verifiable proofs of vulnerability; they prove that an application is vulnerable and they can be independently and inexpensively verified by any host. Hosts use SCAs to analyze the vulnerable application and generate filters or patches to prevent infection.

We show that it is possible to implement general host-based detection engines, for example, we propose a dynamic dataflow analysis technique that can detect attempts to exploit a broad class of vulnerabilities without false negatives and without requiring access to application source code. Additionally, we describe a scalable overlay for broadcasting SCAs that ensures an SCA reaches all hosts with high probability even if the worm is detected by a small number of hosts. This is important to reduce the probability of false negatives; it enables a large-scale collaborative detection system that is hard to attack and allows different hosts to run different detection engines to spread detection load and increase detection coverage. Our preliminary experimental results show that Vigilante can effectively contain fast spreading worms that exploit unknown vulnerabilities.

Section 2 discusses the limitations of the network centric approach and Section 3 discusses how they are overcome by Vigilante. Section 4 describes our implementation. Section 5 presents some preliminary results and Section 6 concludes.

2. NETWORK CENTRIC CONTAINMENT

Containment systems based on *content signatures* assume that the contents of worm messages match a signature, for example, that messages contain a particular byte string. Worm signatures have traditionally been generated by humans but

[1]We will use the term polymorphic to refer to polymorphic, metamorphic and oligomorphic worms [25]

there are several recent proposals to generate signatures automatically [13, 11, 21]. These systems generate signatures for unknown worms by identifying common byte strings in suspicious network flows. They differ in the way they classify network flows as suspicious but they all generate signatures that are byte strings. They can use systems like Cisco's NBAR [24], Bro [18], or Snort [19] to block flows that contain the signature before they reach vulnerable hosts.

Automatic worm containment systems based on byte-string signatures are unable to contain polymorphic worms. These worms change the content of their probe messages constantly by using techniques such as encryption and code obfuscation [25]. Since there is little invariant content across different messages, any signature will be either too long or too short. Signatures that are too long cannot contain worm traffic and signatures that are too short will block normal traffic. Worm writers can use readily available tools [26] to create polymorphic worms. These containment systems are also prone to false positives, for example, an attacker can block normal traffic by generating traffic that is classified as suspicious and contains a byte string that is common in normal traffic. False positives are also possible in the absence of attacks [21]. This is a major barrier to automation; these systems are likely to require human intervention to whitelist signatures that are common in normal traffic.

Another approach to contain worms is based on blocking or rate limiting traffic from hosts that exhibit *abnormal communication patterns*. The rationale is that infected hosts must propagate the worm by communicating with other hosts. Worms that propagate fast must initiate connections to other hosts at a high rate. Snort [19] and Network Security Monitor [8] are based on this observation; they detect worm traffic by monitoring the rate at which unique destination addresses are contacted and block the sender. These approaches can generate false positives and worms that initiate connections just below the rate limit can escape containment.

Other systems are based on the observation that worms that scan random addresses to find vulnerable hosts are likely to target many invalid addresses. For example, Bro [18] uses a configurable threshold on the number of failed connections and Weaver [29] uses a threshold on the ratio of failed to successful connections. Traffic from hosts that exceed these thresholds is blocked. These systems are also prone to false positives and false negatives. For example, an attacker can perform scanning with a fake source address to block traffic from that address. Worms can also evade containment by using techniques that do not cause connection failures; topological, meta-server, passive and target-list worms [28] all use non-scanning techniques.

These network centric worm containment systems are effective against previous worms. However, worm writers can easily increase the sophistication of worms to evade these containment systems.

3. VIGILANTE'S DESIGN

The host centric approach can gather detailed information about vulnerabilities by analyzing the worm infection process inside applications running at hosts. It can avoid the limitations of the network centric approach because it does not rely on the characteristics of the traffic induced by worms, for example, it can detect polymorphic worms and worms that disguise as normal traffic.

It is easy to avoid false positives in host-based worm detection but it can be prone to false negatives. For example, due to narrow coverage all buffer overflow detection engines studied in [30] were unable to detect certain variants of buffer overflow attacks. It is possible to implement host-based detection engines [10, 1] with broad coverage that detect infection attempts with low false negatives and virtually no false positives but the overhead of detection usually grows with the degree of coverage. It would be possible to run these expensive engines continuously on a few dedicated hosts but this centralization would result in false negatives because worms can simply avoid the detectors during propagation.

Vigilante relies on a large scale collaborative worm detection architecture to overcome these problems: every host can be a detector and detectors broadcast an alert to other hosts when they detect a worm outbreak. The architecture allows different hosts to run different detection engines [30, 3, 12, 10, 7, 1] to increase detection coverage and it enables detectors to run expensive engines with broad coverage because it spreads detection load. For example, a host that does not normally run a database server can run an expensive engine to detect attempts to exploit vulnerabilities in a honeypot version of the database server; it will incur little or no overhead for doing so but a production database server would incur an unacceptable overhead. Another alternative is for hosts to run expensive detection engines when they are idle or even to use dedicated hosts. Additionally, this decentralized architecture makes it hard for attackers to evade detection and offers no centralized target for attacks.

Cooperation is important for detection but it is unreasonable to assume that all detectors are trusted in a large scale system. Vigilante introduces self-certifying alerts (SCAs) to eliminate the need for trusting detectors. SCAs are machine-verifiable proofs of vulnerability; they prove that an application is vulnerable. Any host can verify an SCA by using information in the SCA to reproduce the infection process. While the detection mechanisms can be computationally expensive, verification is inexpensive. By decoupling vulnerability detection from verification, SCAs allow a large number of hosts to cooperate in worm detection without trusting each other.

Worm alerts must be broadcast to all hosts that may be running the vulnerable program. For the cooperative worm detection architecture to be effective, the broadcast mechanism must ensure that the SCA reaches all hosts before they become infected with high probability even if the worm is detected by a small number of hosts. It must also withstand denial of service attacks aimed at blocking the propagation of alerts. Anti-virus vendors use centralized services to distribute virus signatures but these are vulnerable to targetted denial of service attacks and it is unclear whether they would withstand the load generated by an acute worm outbreak. Vigilante uses a flooding protocol on a secure overlay [2] that is scalable and resilient to attacks. Other alert distribution mechanisms are possible, for example, IP multicast could be used inside corporate networks but it is not widely deployed on the Internet.

Hosts must protect themselves from infection after receiving an alert but, to prevent denial of service attacks, they do not invest any effort in generating protection mechanisms before they verify the SCA (which is inexpensive). SCAs are vulnerability-centric; they identify a software vulnerability rather than a particular worm. Knowing the vulnerability allows hosts to generate local countermeasures to protect themselves from all worms exploiting that vulnerability. One simple protection is to stop the vulnerable application, which is extreme but may be justifiable in some safety critical settings. A better option is to use the information in the SCA to analyse the vulnerable application and generate patches or filters to prevent infection without stopping the application.

Some concurrent work [20] has proposed a related host centric approach to automate worm containment. But it proposes a different architecture where each organization runs a trusted central service that generates patches automatically using a set of heuristics to modify vulnerable source code. We believe that Vigilante's architecture is more resilient to attack because it is fully decentralized. Since all hosts cooperate in detection and all hosts can protect themselves from infection, there are no preferential targets for attack. Additionally, Vigilante's architecture increases detection coverage because it spreads detection load and allows hosts to run diverse detection engines. The key enabling concept for Vigilante's architecture is self-certifying alerts, which is missing in [20].

4. VIGILANTE'S IMPLEMENTATION

This section discusses our initial implementation of Vigilante and some variations we plan to explore.

4.1 Self-Certifying Alerts

Self-certifying alerts identify a software vulnerability and include information that allows recipients to efficiently check the authenticity of the vulnerability claim. The current implementation of Vigilante generates an SCA by logging non-deterministic events that cause a program to reach a disallowed state and it verifies SCAs by replaying events and checking if the program reaches the disallowed state.

We model the execution of a program as a piecewise deterministic process [6]. The execution is a sequence of intervals, each starting with a non-deterministic event (e.g. receiving a message). Since execution within an interval is deterministic, logging all non-deterministic events enables a complete replay of the execution. Techniques to log and replay non-deterministic events are available from the fault tolerance literature [6, 5]. Replaying the execution that exhibits the vulnerable behaviour allows the node receiving the alert to check its authenticity.

It is important to note that the sequence of events in an SCA does not need to match the events logged during the attack. The host that prepares the SCA can remove events that are not necessary to reach the disallowed state and it can replace the worm code by something inocuous. Since worms are likely to exploit vulnerabilities that do not require a long interaction with the vulnerable program, the sequence of non-deterministic events that needs to be included in SCAs is likely to be short. For many previous worms, a single receive event is sufficient.

Since many worms exploit vulnerabilities that provide an attacker with arbitrary control over the execution of a program, we use an SCA for this type of vulnerability as a running example. We call this type of vulnerability *Arbitrary Execution Control* (AEC).

Any self-certifying alert contains: an identification of the vulnerable program, a vulnerability type, an event list, and optional verification hints. This optional information facilitates checking, for example, the SCA for an AEC specifies where the value that will be loaded into the program counter is in the list of non-deterministic events, e.g., in which message and at which offset.

4.2 Detection

There are several general detection mechanisms [10, 1] that hosts can deploy in detectors. We describe a novel mechanism to detect Arbitrary Execution Control vulnerabilities that uses dynamic data flow analysis.

Many worms inject code into a vulnerable program and force it to execute that code. Another common attack mechanism is to control the execution of the vulnerable program remotely without injecting any new code, for example, forcing a program to call the system() function in the C runtime. We use dynamic data flow to detect attempts to exploit these AEC vulnerabilities. The idea is to track the flow of data received in input operations (e.g. data received from network connections) and block *(i)* any execution of that data and *(ii)* any loads of that data into the program counter. This prevents execution of remotely loaded code and remote execution control.

Our current implementation uses binary re-writing at load time to implement this dynamic dataflow analysis. We instrument every control transfer instruction (e.g. RET, CALL, JMP on x86 CPUs) and every data-movement instruction (e.g. MOV, MOVS, PUSH, POP on x86 CPUs) 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. The dynamic dataflow algorithm is very simple: whenever an instruction that moves data from a destination to a source is executed, the destination becomes dirty if the source is dirty or it becomes clean otherwise. Whenever an input operation is performed (e.g. receiving data from a network connection), the memory locations where the data is written are marked dirty. The instrumented control flow instructions signal an infection attempt when dirty data is about to be executed or loaded into the program counter.

Dynamic dataflow analysis can prevent problems with previous detection tools [30] because it has broader coverage. For example, tools that rely on detecting writes to the return address in the stack are unable to detect attacks that overwrite a function pointer. Dynamic dataflow analysis does not rely on detecting overwrite of any specific data structure. Furthermore, it does not require access to the source code and it works with self-modifying and dynamically generated code.

There are some attacks that dynamic dataflow analysis cannot detect, for example, attacks that exploit backdoors in programs or weak passwords. We are currently extending the implementation to detect attacks that overwrite the arguments of system calls. The idea is to block invocation of dangerous system calls with dirty arguments that have not been checked by the program. This approach is superior to wrapping approaches that compare system call arguments with known bad values because it is more general.

Hardware implementations of mechanisms similar to dynamic dataflow analysis [23, 4] have been proposed concurrently with our work. Vigilante's cooperative detection architecture enables a software-only implementation because it spreads detection load. Additionally, we exploit the flexibility of software to extend the analysis to detect attacks that overwrite system call arguments and to aid in generating SCAs (as described in the next section).

4.3 Alert Generation

When a vulnerability is detected, the host generates an SCA: it generates the summarized event list, removes any information that is not essential to trigger the vulnerability from the events (e.g., the worm code) and generates additional hints to aid verification.

As a concrete example, we describe the generation of an SCA for an AEC using dynamic dataflow analysis. The detector logs non-deterministic events and it intercepts execution when there is an infection attempt (just before the worm code gains control as described before). The detector starts by determining the event in the log that is the source of the dirty address that was about to be executed or loaded into the program counter. Searching through all the events in the log for the specific byte pattern in the address can lead to false matches. Instead, we augment the dynamic data flow analysis to track the origin of dirty data; rather than using a single bit to indicate if a memory location or register is dirty, we use an integer that identifies the input event where dirty data came from. Removing unnecessary data from the event that triggers the vulnerability can be achieved by verifying which portions of event data determine the execution path that triggers the vulnerability.

To reduce the number of events in the alert, we can replay the execution with an increasingly larger suffix of the log and check for the error condition. This strategy is effective for current worms because they trigger vulnerabilities with the last few packets received over a network connection. Further research is required to analyze potential attacks against this alert generation algorithm and to devise additional alert generation algorithms.

4.4 Alert Verification

Verifying an SCA entails replaying the execution that exhibits the vulnerable behaviour. The process of verification is much more efficient than detection and generation of SCAs. For example to verify an AEC alert, a host will:

1. load the vulnerable program as a suspended process
2. load a `Verified` function into the suspended process' address space
3. use the verification hint in the SCA to locate the event and offset of the illegal address identified during detection and replace the illegal address by the address of the `Verified` function
4. replay the execution to force the vulnerable program to jump to the `Verified` function.

It is important that the verification process be run inside a sandboxing environment [5] so that any possible malicious side effects are neutralized. We are currently exploring formal mechanisms similar to Proof-Carrying Code [17] to verify SCAs. The aim is to express the vulnerability as a logic formula and to verify that the preconditions described in the SCA imply the vulnerability. This could allow SCA verification without running the vulnerable program in a sandbox.

4.5 Resilient Diffusion of Alerts

After an SCA has been created, it needs to be rapidly and resiliently distributed to all other nodes that are running the vulnerable program. We are currently evaluating a flooding protocol to distribute SCAs on a secure structured overlay [2]. Hosts in Vigilante are assigned a certified random *Id* by a certification authority to prevent attackers from choosing their identifiers or obtaining many identifiers. Additionally, the identifier of a host determines its set of neighbors in the overlay, which prevents attackers from increasing the fraction of hosts that point to them. Vigilante disseminates an SCA by forwarding it over all overlay neighbor links.

Vigilante imposes some additional requirements on the overlay: *(i)* we need to protect the overlay against denial of service, and *(ii)* worms should be prevented from propagating over the overlay. We protect against denial of service by bounding the rate at which a host may insert messages into the overlay through each of its neighbor links. This is effective because the constraints on neighbor identifiers make it hard for an attacker to change overlay neighbors. We limit membership information leakage by ensuring that Vigilante runs a dedicated overlay and by exploiting the constraints on neighbor identifiers. When node X requests information from node Y, node Y only returns information to node X that it knows node X requires to maintain the overlay. Additionally, it is important to avoid exposing the identities of all overlay neighbors when a host is compromised. This can be achieved, for example, by running the overlay inside the

```
mov al,byte ptr [netbuf]        //move first byte into AL
mov cl,0x31                     //move 0x31 into CL
cmp al,cl                       //compare AL to CL
jne out                         //jump if not equal
xor eax,eax                     //move 0x0 into EAX
loop:
mov byte ptr [esp+eax+4],cl     //move byte into
                                //stack-based buffer
mov cl,byte ptr [eax+netbuf+1]//move next byte into CL
inc eax                         //increment EAX
test cl,cl                      //test if CL is equal to 0x0
jne loop                        //jump if not equal
out:
```

Figure 1: Vulnerable code.

operating system kernel, a virtual machine monitor (VMM) or a hardware chip depending on the level of protection required.

4.6 Local countermeasures

Upon receiving an SCA, hosts can take local action to protect themselves. Hosts can generate patches that correct low-level coding defects or simply stop vulnerable applications. However, our preferred approach is for hosts to generate vulnerability-specific filters installed immediately above the network stack of the hosts [27]. These filters can use application state to decide when to drop incoming traffic and they are resilient to polymorphic worms. The host can generate the conditions for a filter by analyzing the vulnerable execution path identified by the SCA; it can check which bytes in a message determine the execution path that leads to infection and which conditions on those bytes are tested in that execution path.

We will use the vulnerable code in Figure 1 to illustrate how filters can be generated. The code starts by comparing the first byte of the message in the network buffer with a constant (0x31). If it matches, the bytes in the network buffer 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.

Vigilante can generate a filter for this vulnerability by running the SCA verification procedure until the dirty data that will be loaded into the program counter or executed is written. During this execution Vigilante tracks how each dirty memory position or register value is computed from the input message and records all the tests performed. For example, after executing the first four instructions Vigilante would determine the condition that the first byte in the message should be equal to 0x31. Similarly, executing the loop would derive conditions on a sequence of bytes in the network buffer being different from zero. Applying a filter with these conditions to incoming messages would not generate false positives and would block all worm variants exploiting this vulnerability.

We are investigating how to generalize this mechanism to produce filters without false negatives in more complex cases by eliminating conditions that are not decisive to reach the vulnerable code and by analyzing alternative paths to the vulnerability.

5. EVALUATION

This section presents results of experiments to evaluate the performance of Vigilante, under realistic scenarios.

Network topology We used a simple packet-level discrete event simulator that supports different network topologies. Our experiments use a transit-stub topology generated using the Georgia Tech topology generator [31]. It has 5050 routers arranged hierarchically, with 10 transit domains at the top level with 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 run the secure version of Pastry [2] and are attached to randomly selected stub routers by a LAN link with a delay of 1ms. The Pastry configuration uses $b = 1$, $l = 32$, since this provides a good balance between performance and overhead for this application.

Detection and Verification We implemented detection with dynamic dataflow analysis using the Nirvana runtime analysis system [14]. Our preliminary tests show that we can detect vulnerabilities such as the ones described in [30]. The performance overhead of our unoptimized implementation is high (approximately a factor of 50). Even with this high overhead, dynamic dataflow analysis can effectively be used to detect worms because Vigilante's architecture spreads detection load. We expect to achieve much lower overheads by optimizing our implementation. We ran preliminary tests of the operational alert verification mechanism using the Microsoft SQL Server database and the Slammer worm. Having prepared a SQL Server process with the injected `verified` routine, the process of verification entails only forcing SQL Server to jump to that routine by replaying the receive event of the message in an SCA. The process takes a few milliseconds.

Worm propagation model We model worm propagation using the epidemic model described in [9]. The model assumes a network of N nodes and an average infection rate of β. If we represent the total number of infected nodes at time t as I_t, the system is described by the following equation: $\frac{dI_t}{dt} = \beta\, I_t(1 - \frac{I_t}{N})$

Containing the Slammer outbreak Figure 2 plots the fraction of hosts that survive an attack by a worm similar to Slammer, as a function of the fraction of hosts that are capable of detecting the worm. We simulate a population of 100 000 vulnerable hosts of which 10 are initially infected. We assume that detectors are randomly placed, since hosts can decide independently which (if any) detection mechanisms to use. We set $\beta = 0.117$, which is believed to approximate Slammers observed behaviour on the Internet [15]. This means that every 8.5 seconds the number of infected machines doubles (Slammer has been described in [15] as the fastest computer worm in history). Whenever a worm probe reaches a detector, it generates an SCA in 5 seconds and then starts its propagation using the Pastry overlay (preliminary experiments with SQL Server show that we can generate SCAs in less than 1 second, assuming the host is other-

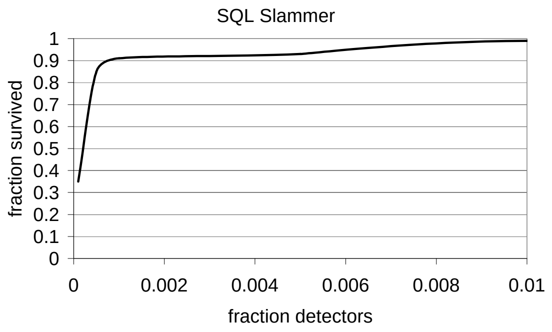

Figure 2: Fraction of survivors for SQL Slammer.

wise idle, but we believe more research is needed to simulate attacks on the SCA generation procedure). The SCA for the vulnerability exploited by Slammer is only 159 bytes. We also assume that even before the worm attack, 10 percent of the hosts are malicious. If a host is malicious or infected it does not propagate SCAs. Each data point in the graph is the worst case of 20 runs, using distinct random placements for the detectors. The graph shows that a very small fraction of detectors (0.001) is enough to contain the worm infection to less than 10 percent of the vulnerable population.

6. CONCLUSIONS

We believe that network centric approaches to automate worm containment have fundamental limitations because there is no information about the vulnerabilities exploited by worms at the network level. Vigilante adopts a host centric approach to automate worm containment that addresses the limitations of the network centric approach by analyzing infection attempts inside applications running at end hosts. Vigilante contains worms using a large scale collaborative architecture to detect worms and to propagate worm alerts. Self-certifying alerts are the fundamental concept that enables this architecture; it eliminates the need for hosts to trust each other and enables hosts to run diverse detection engines and to spread detection load. Hosts can also use SCAs to generate filters or patches that protect themselves from infection. Our preliminary experimental results show that Vigilante can contain very fast spreading worms such as Slammer.

7. REFERENCES

[1] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zov. Randomized instruction set emulation to disrupt binary code injection attacks. In *CCS*, Washington D.C., USA, October 2003.

[2] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Security for structured peer-to-peer overlay networks. In *OSDI*, Boston, USA, December 2002.

[3] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Wadpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic detection and prevention of buffer-overrun attacks. In *7th USENIX Security Symposium*, San Antonio, USA, January 1998.

[4] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *MICRO-37*, Portland, OR, USA, December 2004.

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

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

[7] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *SNDSS*, pages 191–206, San Diego, USA, February 2003.

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

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

[10] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS*, Washington D.C., USA, October 2003.

[11] H. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, USA, August 2004.

[12] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, San Francisco, USA, August 2002.

[13] C. Kreibich and J. Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *HotNets-II*, Cambridge, USA, November 2003.

[14] Microsoft. Nirvana: A runtime analysis infrastructure. `http://research.microsoft.com/bit/`.

[15] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

[16] David Moore, Colleen Shannon, Geoffrey Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *Infocom*, San Francisco, USA, April 2003.

[17] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *OSDI*, Seattle, USA, October 1996.

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

[19] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th Conference on Systems Administration*, Seattle, USA, November 1999.

[20] Stelios Sidiroglou and Angelos D. Keromytis. Countering network worms through automatic patch generation. *IEEE Security and Privacy*, 2005.

[21] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. The earlybird system for real-time detection of unknown worms. Technical Report CS2003-0761, University of California, San Diego, August 2003.

[22] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the internet in your spare time. In *USENIX Security Symposium 2002*, San Francisco, USA, August 2002.

[23] G. Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS XI*, Boston, MA, USA, October 2004.

[24] Cisco Systems. Network-based application recognition.

[25] Peter Szor and Peter Ferrie. Hunting for metamorphic. In *Virus Bulletin Conference*, Prague, September 2001.

[26] theo detristan, tyll ulenspiegel, yann_malcom, and mynheer superbus von underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack*, 11(49), August 2003.

[27] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM*, Portland, USA, August 2004.

[28] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham. A taxonomy of computer worms. In *The First ACM Workshop on Rapid Malcode (WORM)*, 2003.

[29] N. Weaver, S. Staniford, and V. Paxson. Very fast containment of scanning worms. In *Proceedings of the 13th USENIX Security Symposium*, San Diego, USA, August 2004.

[30] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *NDSS*, pages 149–162, San Diego, California, February 2003.

[31] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *INFOCOM96*, San Francisco, California, 1996.