A Network Worm Vaccine Architecture

Stelios Sidiroglou
Columbia University
stelios@cs.columbia.edu

Angelos D. Keromytis
Columbia University
angelos@cs.columbia.edu

Abstract

The ability of worms to spread at rates that effectively preclude human-directed reaction has elevated them to a first-class security threat to distributed systems. We present the first reaction mechanism that seeks to automatically patch vulnerable software. Our system employs a collection of sensors that detect and capture potential worm infection vectors. We automatically test the effects of these vectors on appropriately-instrumented sandboxed instances of the targeted application, trying to identify the exploited software weakness. Our heuristics allow us to automatically generate patches that can protect against certain classes of attack, and test the resistance of the patched application against the infection vector. We describe our system architecture, discuss the various components, and propose directions for future research.

1 Introduction

Recent incidents [4, 5] have demonstrated the ability of self-propagating code, also known as "network worms" [31, 9], to infect large numbers of hosts, exploiting vulnerabilities in the largely homogeneous deployed software base [6, 41]. Even when the worm carries no malicious payload, the direct cost of recovering from the side effects of an infection epidemic can be tremendous [1]. Thus, countering worms has recently become the focus of increased research. Analysis of the Code Red worm [25, 32] has shown that even modest levels of immunization and dynamic counter-measures can considerably limit the infection rate of a worm. Other simulations confirm this result [37]. However, these epidemics have also demonstrated the ability of worms to achieve extremely high infection rates. This implies that a hypothetical reaction system must not, and in some cases cannot, depend on human intervention for containment [34, 26].

Most of the research for countering worms has focused on prevention, with little attention paid to reaction systems. Work on application protection (prevention) has produced solutions that either have considerable impact on performance or are inadequate. *We propose a first-reaction mechanism that tries to automatically patch vulnerable software, thus circumventing the need for human intervention in time-critical infection containment.* In our system, the issue of performance is of diminished importance since the primary use of the protection mechanisms is to identify the source of weakness in the application.

The approach presented here employs a combination of techniques such as the use of honeypots, dynamic code analysis, auto-patching, sandboxing, and software updates. The ability to use these techniques is contingent upon a number of (we believe) realistic assumptions. Dynamic analysis relies on the assumption that we expect to tackle known classes of attacks. The generation of patches depends on source availability, although techniques such as binary rewriting may be applicable. The ability to provide a fix implies a trusted computing base (TCB) of some form, which we envision would be manifest in the form of a virtual machine [3] or an application sandbox [28].

Our architecture can be deployed on a per-network or per-organization basis, reflecting the trust relationships among entities. This can be further dissected to distributed sensors, anomaly detection, and the sandboxed environment. For example, an enterprise network may use a complete instantiation of our architecture while also providing remote-sensor functionality to other organizations.

The benefits presented by our system are the quick reaction to attacks by the automated creation of 'good enough' fixes without any sort of dependence on a central authority, such as a hypothetical Cyber-CDC [34]. Comprehensive security measures (CERT, vendor updates, *etc.*) can be administered at a later time. A limitation of the system is that it is able to counter only known classes of attacks (*e.g.,* buffer-overflow attacks). We feel that this does not severely hinder functionality, as most attacks use already-known techniques. Furthermore, our architecture is easily extensible to accommodate detection and reactive measures against new types of attacks as they become known.

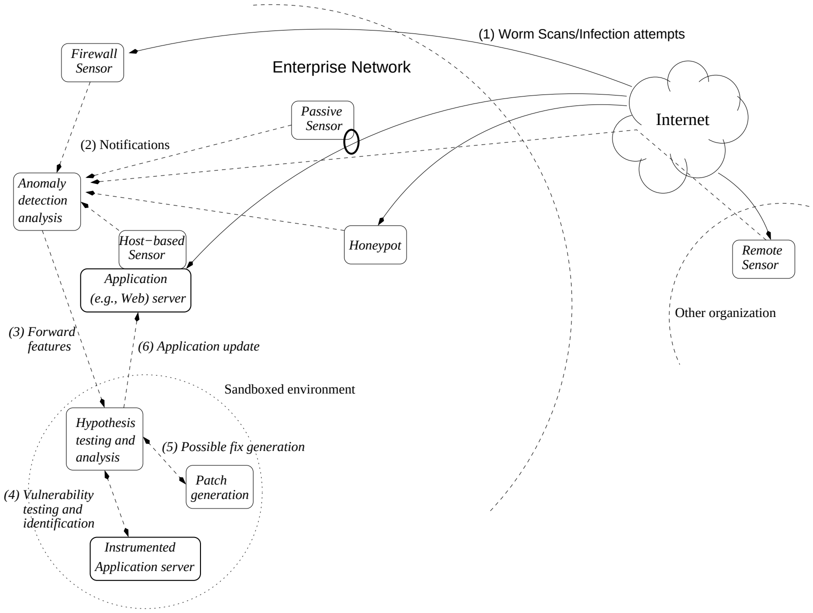
(1) Worm Scans/Infection attempts
Firewall Sensor
Enterprise Network
Internet
Passive Sensor
(2) Notifications
Anomaly detection analysis
Host−based Sensor
Honeypot
Remote Sensor
Application (e.g., Web) server
Other organization
(3) Forward features
(6) Application update
Sandboxed environment
Hypothesis testing and analysis
(5) Possible fix generation
Patch generation
(4) Vulnerability testing and identification
Instrumented Application server

Figure 1. Worm vaccination architecture: sensors deployed at various locations in the network detect a potential worm (1), notify an analysis engine (2) which forwards the infection vector and relevant information to a protected environment (3). The potential infection vector is tested against an appropriately-instrumented version of the targeted application, identifying the vulnerability (4). Several software patches are generated and tested using several different heuristics (5). If one of them is not susceptible to the infection and does not impact functionality, the main application server is updated (6).

2 System Architecture

Our architecture, shown in Figure 1, makes use of five types of components: a set of worm-detection sensors, an anomaly-detection engine, a sandboxed environment running appropriately-instrumented versions of the applications used in the enterprise network (*e.g.,* Apache web server, SQL database server, *etc.*), an analysis and patch-generation engine, and a software update component. We describe each of these components in turn.

The worm-detection sensors are responsible for detecting potential worm probes and, more importantly, infection attempts. Several types of sensors may be employed concurrently:

- Host-based sensors, monitoring the behavior of deployed applications and servers.

- Passive sensors on the corporate firewall or on independent boxes, eavesdropping on traffic to and from the servers.

- Special-purpose honeypots, that simulate the behavior of the target application and capture any communication.

- Other types of sensors, including sensors run by other entities (more on this in Section 3).

Any combination of the above sensors can be used simultaneously, although we believe honeypot servers are the most promising, since worms cannot distinguish between real and fake servers in their probes. These sensors can communicate with each other and with a central server, which can correlate events from independent sensors and determine potential infection vectors (*e.g.,* the data on an HTTP request, as seen by a honeypot). In general, we assume that a worm can *somehow* be detected. We believe our assumption to be pragmatic, in that any likely solution to the worm problem is predicated upon this.

The potential infection vector (*i.e.,* the byte stream which, when “fed” to the target application, will cause an instance of the worm to appear on the target system) is forwarded to a sandboxed environment, which runs appropriately-instrumented instances of the applications we are interested in protecting (*e.g.,* Apache or IIS server). The instrumentation can be fairly invasive in terms of performance, since we only use the server for clean-room testing. In its most powerful form, a full-blown machine emulator [30] can be used to determine whether the application has been subverted. Other potential instrumentation includes light-weight virtual machines [12, 16, 38], dynamic analysis/sandboxing tools [8, 20, 19], or mechanisms such as MemGuard [11]. These mechanisms are generally not used for application protection due to their considerable impact on performance. In our system, this drawback is not of particular importance because we only use these mechanisms to identify as accurately as possible the source of weakness in the application. For example, MemGuard [11] or *libverify* [8] can both identify the specific buffer and function that is exploited in a buffer-overflow attack. Alternatively, when running under a simulator, we can detect when execution has shifted to the stack, heap, or some other unexpected location, such as an unused library function.

The more invasive the instrumentation, the higher the likelihood of detecting subversion and identifying the source of the vulnerability. Although the analysis step can only identify known classes of attack (*e.g.,* a stack-based buffer overflow [7]), even if it can detect anomalous behavior, new classes of attack (*e.g.,* heap-based overflows [23]) appear less often than exploits of known attack types do.

Armed with knowledge of the vulnerability, we can automatically patch the program. Generally, program analysis is an impossible problem (*Halting* problem). However, there are a few fixes that may mitigate the effects of an attack. Some potential fixes to buffer-overflow attacks include:

- Moving the offending buffer to the heap, by dynamically allocating the buffer upon entering the function and freeing it at all exit points. Furthermore, we can increase the size of the allocated buffer to be larger than the size of the infection vector, thus protecting the application from even crashing. Finally, we can use a version of *malloc()* that allocates two additional write-protected pages that bracket the target buffer. Any buffer overflow or underflow will cause the process to receive a Segmentation Violation signal, which we catch in a signal handler of ours. The signal handler can then *longjmp()* to the code immediately after the routine that caused the buffer overflow.

- For heap-based attacks that overflow buffers in dynamically-allocated objects (in object-oriented languages), we can reorder fields in the object.

- More generally, we can use some minor code-randomization techniques [14] that could shift the vulnerability such that the infection vector no longer works.

- We can add code that recognizes either the attack itself or specific conditions in the stack trace (*e.g.,* a specific sequence of stack records), and returns from the function if it detects these conditions. The former is in some sense equivalent to content filtering, and least likely to work against even mildly polymorphic worms. Generally, this approach appears to be the least promising.

- Finally, we can attempt to "slice-off" some functionality, by immediately returning from mostly-unused code that contains the vulnerability. Especially for large software systems that contain numerous, often untested, features that are not regularly used, this may be the solution with the least impact. We can determine whether a piece of functionality is unused by profiling the real application; if the vulnerability is in an unused section of the application, we can logically remove that part of the functionality (*e.g.,* by an early function-return).

Our architecture makes it possible to easily add new analysis techniques and patch-generation components. To generate the patches, we are experimenting with Stratego [36] and TXL [24], two code-transformation tools.

We can test several patches (potentially even simultaneously, if enough resources are available), until we are satisfied that the application is no longer vulnerable to the specific exploit. To ensure that the patched version will continue to function, regression testing can be used to determine what functionality (if any) has been lost. The test suite is generated by the administrator in advance, and should reflect a typical workload of the application, exercising all critical aspects (*e.g.,* performing purchasing transactions). Naturally, one possibility is that no heuristic will work, in which case it is not possible to automatically fix the application and other measures have to be used.

Once we have a worm-resistant version of the application, we must instantiate it on the server. Thus, the last component of our architecture is a server-based monitor. To achieve this, we can either use a virtual-machine approach or assume that the target application is somehow sandboxed and implement the monitor as a regular process residing outside that sandbox. The monitor receives the new version of the application, terminates the running instance (first attempting a graceful termination), replaces the executable with the new one, and restarts the server.

3 Discussion

Following the description of our architecture, there are several other issues that need to be discussed.

The main challenges in our approach are, (1) determining the nature of the attack (*e.g.,* buffer overflow) and identifying the likely software flaws that permit the exploit and, (2) reliably repairing the software. Obviously, our approach can only fix attacks it already "knows" about, *e.g.,* stack or heap-based buffer overflows. This knowledge manifests itself through the debugging and instrumentation of the sandboxed version of the application. Currently, we use ProPolice [13] to identify the likely functions and buffers that lead to the overflow condition. More powerful analysis tools [30, 19, 8] can be easily employed in our architecture to catch more sophisticated code-injection attacks. One advantage of our approach is that the performance implications of such mechanisms are not relevant: an order of magnitude or more slow-down of the instrumented application is acceptable, since it does not impact the common-case usage. Furthermore, our architecture should be general enough that other classes of attack can be detected, *e.g.*, email worms, although we have not yet investigated this.

As we mentioned already, repairability is impossible to guarantee, as the general problem can be reduced to the Halting Problem. Our heuristics allow us to generate potential fixes for several classes of buffer overflows using code-to-code transformations [24], and test them in a clean-room environment. Further research is necessary in the direction of automated software recovery in order to develop better repair mechanisms. Interestingly, our architecture could be used to automatically fix any type of software fault, such as invalid memory dereference, by plugging in the appropriate repair module. When it is impossible to automatically obtain a software fix, we can use content-filtering as in [29] to temporarily protect the service. The possibility of combining the two techniques is a topic of future research.

Our system assumes that the source code of the instrumented application is available, so patches can be easily generated and tested. When that is not the case, binary-rewriting techniques may be applicable, at considerably higher complexity. Instrumentation of the application also becomes correspondingly more difficult under some schemes. One intriguing possibility is that vendors ship two versions of their applications, a "regular" and an "instrumented" one; the latter would provide a standardized set of hooks that would allow a general monitoring module to exercise oversight.

The authors of [34] envision a Cyber "Center for Disease Control" (CCDC) for identifying outbreaks, rapidly analyzing pathogens, fighting the infection, and proactively devising methods of detecting and resisting future attacks. However, it seems unlikely that there would ever be wide acceptance of an entity trusted to arbitrarily patch software running on any user's system[1]. Furthermore, fixes would still be need to be handcrafted by humans and thus arrive too late to help in worm containement. In our scheme, such a CCDC would play the role of a real-time alert-coordination and distribution system. Individual enterprises would be able to independently confirm the validity of a reported weakness and create their own fixes in a decentralized manner, thereby minimizing the trust they would have to place to the CCDC.

Note that although we speculate the deployment of such a system in every medium to large-size enterprise network, there is nothing to preclude pooling of resources across multiple, mutually trusted, organizations. In particular, a managed-security company could provide a quick-fix service to its clients, by using sensors in every client's location and generating patches in a centralized facility. The fixes would then be pushed to all clients.

One concern in our system is the possibility of "gaming" by attackers, causing instability and unnecessary software updates. One interesting attack would be to cause oscillation between versions of the software that are alternatively vulnerable to different attacks. Although this may be theoretically possible, we cannot think of a suitable example. Such attack capabilities are limited by the fact that the system can test the patching results against both current and previous (but still pending, *i.e.*, not "officially" fixed by an administrator-applied patch) attacks. Furthermore, we assume that the various system components are appropriately protected against subversion, *i.e.*, the clean-room environment is firewalled, the communication between the various components is integrity-protected using TLS/SSL or IPsec.

If a sensor is subverted and used to generate false alarms, event correlation will reveal the anomalous behavior. In any case, the sensor can at best only mount a denial of service attack against the patching mechanism, by causing many hypotheses to be tested. Again, such anomalous behavior is easy to detect and take into consideration without impacting either the protected services or the patching mechanism.

Another way to attack our architecture involves denying the communication between the correlation engine, the sensors, and the sandbox through a denial of service attack. Such an attack may in fact be a by-product of a worm's aggressive propagation, as was the case with the SQL worm [6]. Fortunately, it should be possible to ingress-filter the ports used for these communications, making it very difficult to mount such an attack from an external network. To protect communication with remote sensors, we can use the SOS architecture [18].

4 Related Work

In [34], the authors describe the risk to the Internet due to the ability of attackers to quickly gain control of vast numbers of hosts. They argue that controlling a million hosts can have catastrophic results because of the potential to launch distributed denial of service (DDoS) attacks and potential access to sensitive information that is present on those hosts. Their analysis shows how quickly attackers can compromise hosts using "dumb" worms and how "better" worms can spread even faster.

Since the first Internet-wide worm [33], considerable effort has gone into preventing worms from exploiting com-

[1] Although certain software vendors are perhaps near that point.

mon software vulnerabilities by using the compiler to inject run-time safety checks into applications (*e.g.,* [11]), safe languages and APIs, and static or dynamic [20] analysis tools. Several shortcomings are associated with these tools: some are difficult to use, have a steep learning curve (especially for new languages), or impose significant performance overheads. Furthermore, they are not always successful in protecting applications against old [10, 39] or new [23] classes of attacks. Finally, they require proactiveness from deadline-driven application developers.

Another approach has been that of containment of infected applications, exemplified by the "sandboxing" paradigm (*e.g.,* [17]). Unfortunately, even when such systems are successful in containing the virus [15], they do not prevent further propagation or ensure continued service availability [21]. Furthermore, there is often a significant performance overhead associated with their use.

Most of the existing anti-virus techniques use a simple signature scanning approach to locate threats. As new viruses are created, so do virus signatures. Smarter virus writers use more creative techniques (*e.g.,* polymorphic viruses) to avoid detection. In response detection mechanisms become ever more elaborate. This has led to coevolution [27], an ever-escalating arms race between virus writers and anti-virus developers.

The HACQIT architecture [29] uses various sensors to detect new types of attacks against secure servers, access to which is limited to small numbers of users at a time. Any deviation from expected or known behavior results in the possibly subverted server to be taken off-line. Similar to our approach, a sandboxed instance of the server is used to conduct "clean room" analysis, comparing the outputs from two different implementations of the service (in their prototype, the Microsoft IIS and Apache web servers were used to provide application diversity). Machine-learning techniques are used to generalize attack features from observed instances of the attack. Content-based filtering is then used, either at the firewall or the end host, to block inputs that may have resulted in attacks, and the infected servers are restarted. Due to the feature-generalization approach, trivial variants of the attack will also be caught by the filter. [35] takes a roughly similar approach, although filtering is done based on port numbers, which can affect service availability. Cisco's Network-Based Application Recognition (NBAR) [2] allows routers to block TCP sessions based on the presence of specific strings in the TCP stream. This feature was used to block Code-Red probes, without affecting regular web-server access.

Code-Red inspired several countermeasure technologies. La Brea [22] attempts to slow the growth of TCP-based worms by accepting connections and then blocking on them indefinitely, causing the corresponding worm thread to block. Unfortunately, worms can avoid this mechanisms by probing and infecting asynchronously. Under the connection-throttling approach [40], each host restricts the rate at which connections may be initiated. If universally adopted, such an approach would reduce the spreading rate of a worm by up to an order of magnitude, without affecting legitimate communications.

5 Conclusion

We presented an architecture for countering self-propagating code (worms) through automatic software-patch generation. Our architecture uses a set of sensors to detect potential infection vectors, and uses a clean-room (sandboxed) environment running appropriately-instrumented instances of the applications used in the enterprise network to test potential fixes. To generate the fixes, we use code-transformation tools to implement several heuristics that counter specific buffer-overflow instances. We iterate until we create a version of the application that is both resistant to the worm and meets certain minimal-functionality criteria, embodied in an regression test suite created in advance by the system administrator.

Our proposed system allows quick, automated reaction to worms, thereby simultaneously increasing service availability and potentially decreasing the worm's infection rate [41]. The emphasis is on generating a quick fix; a more comprehensive patch can be applied at a later point in time, allowing for secondary reaction at a human time-scale. Although our system cannot fix all types of attacks, experimentation with our preliminary heuristics is promising.

Future research is needed in better analysis and patch-generation tools. We believe that our architecture, combined with other approaches such as content-filtering, can significantly reduce the impact of worms in service availability and network stability.

References

[1] 2001 Economic Impact of Malicious Code Attacks. `http://www.computereconomics.com/cei/press/pr92101.html`.

[2] Using Network-Based Application Recognition and Access Control Lists for Blocking the "Code Red" Worm at Network Ingress Points. Technical report, Cisco Systems, Inc.

[3] VMWare Emulator. `http://www.vmware.com/`.

[4] CERT Advisory CA-2001-19: 'Code Red' Worm Exploiting Buffer Overflow in IIS Indexing Service DLL. `http://www.cert.org/advisories/CA-2001-19.html`, July 2001.

[5] Cert Advisory CA-2003-04: MS-SQL Server Worm. `http://www.cert.org/advisories/CA-2003-04.html`, January 2003.

[6] The Spread of the Sapphire/Slammer Worm. `http://www.silicondefense.com/research/worms/slammer.php`, February 2003.

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

[8] A. Baratloo, N. Singh, and T. Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[9] J. Brunner. *The Shockwave Rider*. Del Rey Books, Canada, 1975.

[10] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 5(56), May 2000.

[11] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[12] 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 *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[13] J. Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp/`, June 2000.

[14] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the 6th HotOS Workshop*, 1997.

[15] T. Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS)*, pages 163–176, February 2003.

[16] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS)*, pages 191–206, February 2003.

[17] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Procedings of the 1996 USENIX Annual Technical Conference*, 1996.

[18] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proceedings of ACM SIGCOMM*, pages 61–72, August 2002.

[19] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–205, August 2002.

[20] K. Lhee and S. J. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–90, August 2002.

[21] M.-J. Lin, A. Ricciardi, and K. Marzullo. A New Model for Availability in the Face of Self-Propagating Attacks. In *Proceedings of the New Security Paradigms Workshop*, November 1998.

[22] T. Liston. Welcome To My Tarpit: The Tactical and Strategic Use of LaBrea. Technical report, 2001.

[23] M. Conover and w00w00 Security Team. w00w00 on heap overflows. `http://www.w00w00.org/files/articles/heaptut.txt`, January 1999.

[24] A. J. Malton. The Denotational Semantics of a Functional Tree-Manipulation Language. *Computer Languages*, 19(3):157–168, 1993.

[25] D. Moore, C. Shanning, and K. Claffy. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of the 2nd Internet Measurement Workshop (IMW)*, pages 273–284, November 2002.

[26] D. Moore, C. Shannon, G. Voelker, and S. Savage. Internet Quarantine: Requirements for Containing Self-Propagating Code. In *Proceedings of the IEEE Infocom Conference*, April 2003.

[27] C. Nachenberg. Computer Virus - Coevolution. *Communications of the ACM*, 50(1):46–51, 1997.

[28] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[29] J. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich. The Design and Implementation of an Intrusion Tolerant System. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2002.

[30] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *Modeling and Computer Simulation*, 7(1):78–103, 1997.

[31] J. Shoch and J. Hupp. The "worm" programs – early experiments with a distributed computation. *Communications of the ACM*, 22(3):172–180, March 1982.

[32] D. Song, R. Malan, and R. Stone. A Snapshot of Global Internet Worm Activity. Technical report, Arbor Networks, November 2001.

[33] E. H. Spafford. The Internet Worm Program: An Analysis. Technical Report CSD-TR-823, Purdue University, 1988.

[34] S. Staniford, V. Paxson, and N. Weaver. How to Own the Internet in Your Spare Time. In *Proceedings of the 11th USENIX Security Symposium*, pages 149–167, August 2002.

[35] T. Toth and C. Kruegel. Connection-history Based Anomaly Detection. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, June 2002.

[36] E. Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. *Lecture Notes in Computer Science*, 2051, 2001.

[37] C. Wang, J. C. Knight, and M. C. Elder. On Computer Viral Infection and the Effect of Immunization. In *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC)*, pages 246–256, 2000.

[38] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[39] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Intrusion Prevention. In *Proceedings of the Symposium on Network and Distributed Systems Security (SNDSS)*, pages 123–130, February 2003.

[40] M. Williamson. Throttling Viruses: Restricting Propagation to Defeat Malicious Mobile Code. Technical Report HPL-2002-172, HP Laboratories Bristol, 2002.

[41] C. C. Zou, W. Gong, and D. Towsley. Code Red Worm Propagation Modeling and Analysis. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 138–147, November 2002.