Countering Network Worms Through Automatic Patch Generation

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 propose an architecture for automatically repairing software flaws that are exploited by **zero-day worms.** *Our approach relies on source code transformations to quickly apply* **automatically-created (and tested) localized patches** *to vulnerable segments of the targeted application. To determine these susceptible portions, we use a sandboxed instance of the application as a "clean room" laboratory that runs in parallel with the production system and exploit the fact that a worm must reveal its infection vector to achieve its goal (i.e., further infection). We believe our approach to be* **the first end-point solution** *to the problem of malicious self-replicating code. The primary benefits of our approach are* (a) *its low impact on application performance,* (b) *its ability to respond to attacks without human intervention, and* (c) *its capacity to deal with zero-day worms (for which no known patches exist). Furthermore, our approach does not depend on a centralized update repository, which can be the target of a concerted attack similar to the Blaster worm. Finally, our approach can also be used to protect against lower intensity attacks, such as intrusion ("hack-in") attempts. To experimentally evaluate the efficacy of our approach, we use our prototype implementation to test a number of applications with known vulnerabilities. Our preliminary results indicate a success rate of 82%, and a maximum repair time of 8.5 seconds.*

1 Introduction

Recent incidents [6, 7, 9, 90] have demonstrated the ability of self-propagating code, also known as "network worms" [91, 19], to infect large numbers of hosts, exploiting vulnerabilities in the largely homogeneous deployed software base [10, 118, 5, 8]. Even when a 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, generally focusing on content-filtering mechanisms combined with large-scale coordination strategies [71, 99, 73, 47].

Despite some promising early results [57], we believe that this approach will be insufficient by itself in the future. We base this primarily on two observations. First, to achieve coverage, such mechanisms are intended for use by routers (*e.g.,* Cisco's NBAR [4]); given the routers' limited budget in terms of processing cycles per packet, even mildly polymorphic worms (mirroring the evolution of polymorphic viruses, more than a decade ago) [100] are likely to evade such filtering [25, 60] or render it impractical [95]. Network-based intrusion detection systems (NIDS) have encountered similar problems, requiring fairly invasive packet processing and queuing at the router or firewall. When placed in the application's critical path, as such filtering mechanisms must, they will have an adverse impact on performance. Second, end-to-end "opportunistic"[1] encryption in the form of TLS/SSL [32] or IPsec [54] is being used by an increasing number of hosts and applications [2]. We believe that it is only a matter of time until worms start using such encrypted channels to cover their tracks. Similar to the case for distributed firewalls [15, 46], these trends argue for an end-point worm-countering mechanism.

[1] By "opportunistic" we mean that client-side, and often server-side, authentication is often not strictly required, as is the case with the majority of web servers or with SMTP over TLS (*e.g.,* sendmail's STARTSSL option).

A preventative approach to the worm problem is the elimination or minimization of remotely-exploitable vulnerabilities, such as buffer overflows. Detecting potential buffer overflows is a very difficult problem, for which only partial solutions exist (*e.g.,* [23, 61]). "Blanket" solutions such as StackGuard or MemGuard [30] typically exhibit at least one of two problems: reduced system performance, and self-induced denial of service (*i.e.,* when an overflow is detected, the only alternative is to terminate the application). Thus, they are inappropriate for high-performance, high-availability environments such as a heavily-used e-commerce web server. An ideal solution would use expensive protection mechanisms only where needed and allow applications to gracefully recover from such attacks.

We propose an end-point first-reaction mechanism that tries to automatically patch vulnerable software by identifying and transforming the code surrounding the exploited software flaw. Briefly, we use instrumented versions of an enterprise's important services (*e.g.,* web server) in a sandboxed environment. This environment is operated *in parallel with* the production servers, and is not used to serve actual requests. Instead, we use it as a "clean room" environment to test the effects of "suspicious" requests, such as potential worm infection vectors. A request that causes a buffer overflow on the production server will have the same effect on the sandboxed version of the application. Appropriate instrumentation allows us to determine the buffers and functions involved in a buffer overflow attack. We then apply several source-code transformation heuristics that aim to contain the buffer overflow. Using the same sandboxed environment, we test the patches against both the infection vectors and a site-specific functionality test-suite, to determine correctness. If successful, we update the production servers with the new version of the targeted program. We are careful to produce localized patches, for which we can be confident that they will not introduce additional instabilities. Note that the patch generation and testing occurs in a completely *decentralized* and *real-time* fashion, without need for a centralized update site, which may become an attack target, as happened with the W32/Blaster worm [9].

Our architecture makes use of several components that have been developed for other purposes. Its novelty lies in the combination of all the components in fixing vulnerable applications without unduly impacting their performance or availability. Our major assumption is that we can extract a worm's infection vector (or, more generally, one instance of it, for polymorphic worms). As we discuss in Section 2, we envision the use of various mechanisms such as honeypots [96, 31], host-based, and network-based intrusion detection sensors. Note that vector-extraction is a necessary precondition to any reactive or filtering-based solution to the worm problem. A secondary assumption is that the source code for the application is available. Although our architecture can use binary rewriting techniques [78], in this paper we focus on source-code transformations. We should also note that several popular server applications are in fact open-source (*e.g.,* Apache [5], Sendmail, MySQL, Bind). Furthermore, although our architecture can be used verbatim to react to lower-intensity "hack-in" attempts, in this paper we focus on the higher-profile (and easier to detect) worm problem.

To determine the effectiveness of our approach, we tested a set of 17 applications vulnerable to buffer overflows, compiled by the Cosak project [3]. We simulated the presence of a worm (even for those applications that were not in fact network services) by triggering the buffer overflow that the worm would exploit to infect the process. Our experiments show that our architecture was able to fix the problem in 82% of all cases. An experiment with a hypothetical vulnerability in the Apache web server showed that the total time to produce a correct and tested fix was 8.3 seconds. This means that the total cycle from detection to updating the real server can be less than 30 seconds. We believe that this is sufficiently fast to suppress most, if not all worm attacks in a completely decentralized manner.

1.1 Paper Organization

The remainder of this paper is organized as follows. We present our system architecture in Section 2, and describe a prototype implementation in Section 3. We evaluate its performance and effectiveness in Section 4. Section 5 discusses potential improvements and open issues, while Section 6 gives an overview of related work.

2 System Architecture

Our architecture, depicted in Figure 1, makes use of five types of components: a set of worm-detection sensors, a correlation 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 now describe each of these components.

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).**

2.1 Worm Detection and Correlation Engine

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 from/to the servers.
- 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 5).

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. Honeypots and other deception-based mechanisms are also effective against hit-list-based worm propagation strategies, assuming they have been in place during the scanning phase [99].

These sensors communicate with each other and with a central server, which correlates events from independent sensors and determines 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 most current solutions to the worm problem depend on a satisfactory answer to the problem of worm detection.

The purpose of the correlation engine is to determine the likelihood of any particular communication being an infection vector (or a manually-launched attack) and to request that the suspicious communication be tested in the sandbox environment. It also maintains a list of fixed exploits and of false positives (communications that have already been determined to be innocuous, or at least ineffective against the targeted application).

2.2 Sandboxed Environment

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 [85] can be used to determine whether the application has been subverted. Other potential instrumentation includes light-weight virtual machines [33, 41, 110], dynamic analysis/sandboxing tools [14, 64, 58], or mechanisms such as MemGuard [30]. These mechanisms are generally not used for application protection due to their considerable impact on performance and the fact that they typically cause the application to "fault", rather than continue operation. In our approach, this drawback is not of particular importance because we only use these mechanisms in the sandboxed environment to identify as accurately as possible the source of weakness in the application. These mechanisms are not used for the production servers. For example, MemGuard [30] or *libverify* [14] can identify both 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 [11]), even if it can detect anomalous behavior, new classes of attack (*e.g.,* heap-based overflows [67]) appear less often than exploits of known attack types. *Note that we are not limited to known exploits and attacks.*

2.3 Patch Generation and Testing

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

- 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 (for fixed-size exploits). 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 (SEGV) signal. This signal is caught by a signal handler we have added to the source code. The signal handler can then *longjmp()* to the code immediately after the routine that caused the buffer overflow. Although this approach could be used in a "blanket" manner (*i.e.,* applied everywhere in the code where a buffer overflow *could* occur, the performance implications would be significant. Instead, we use the worm's infection vector as a hint to locate the potential vulnerability, somewhat simplifying the problem. We give more details in Section 3.

- Use some minor code-randomization techniques [37, 16] that could "move" the vulnerability such that the infection vector no longer works.

- 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 [95]. 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).

We focus on the first approach, as it seems the most promising. We plan to further investigate other heuristics in future research. The patches we introduce are localized, to avoid introducing additional instability to the application. Although it is very difficult, if not impossible, to argue about the correctness of any newly introduced code (whether it was created by a human or an automated process such as ours), we are confident that our patches do not exacerbate the problem because of their minimal scope and the fact that they emulate behavior that could have been introduced automatically by the compiler or some other automated tool during the code authoring or compilation phase. Although this is by no means a proof of correctness, we believe it is a good argument with respect to the safety of the approach.

Our architecture makes it possible to add new analysis techniques and patch-generation components easily. To generate the patches, we employ TXL [68], a language-independent code-transformation tool. We describe its use in more detail in Section 3 and in Appendix A.

We can test several patches (potentially in parallel) 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, a site-specific test suite is 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.

2.4 Application Update

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 [33, 41] or assume that the target application is somehow sandboxed (see Section 6) 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 version, and restarts the server. Another approach is to use dynamic code updating [12] to avoid any application down-time.

3 Implementation

Our prototype implementation is comprised of three components: ProPolice, TXL, and a sandboxed environment. These components interact to identify software vulnerabilities, apply potential patches, and provide a secure environment respectively. In Section 4 we use the implementation to simulate attacks and provide fixes for a sample service application and a list of vulnerable open-source products compiled by the Code Security Analysis Kit (CoSAK) project [3]. Here, we introduce the components and discuss the implementation.

3.1 ProPolice

In order to detect the source of buffer overflow/underflow vulnerabilities, we employ the OpenBSD version of ProPolice [36]. ProPolice will return the names of the function and offending buffer that lead to the overflow condition. This information is then forwarded to a TXL program that attempts a number of heuristics, as discussed in Section 2.

ProPolice is a GCC extension for protecting applications from stack-smashing attacks. Applications written in C and compiled with a ProPolice-enabled version of GCC are automatically protected. The protection is realized by buffer overflow detection and the variable reordering feature to avoid the corruption of pointers. The basic idea of buffer overflow detection comes from the StackGuard system [30]. Its novel features are (1) the reordering of local variables to place buffers after pointers to avoid the corruption of pointers that could be used to further corrupt arbitrary memory locations, (2) the copying of pointers in function arguments to an area preceding local variable buffers to prevent the corruption of pointers that could be used to corrupt arbitrary memory locations further, and (3) the omission of instrumentation code from some functions to decrease the performance overhead.

When a buffer overflow attack is attempted on applications compiled with the ProPolice extensions, the execution of the program is interrupted and the offending function and buffer are reported. When used to protect a service, ProPolice incurs a modest performance overhead, similar to StackGuard's [30]. More importantly, the application

under attack is terminated. While this is more palatable than outright subversion, it is sub-optimal in terms of service availability.

Better mechanisms to use include Valgrind [89] or MemGuard [30]. Although ProPolice was sufficient for our prototype implementation, a fully-functional system would use either of these systems to catch *all* illegal memory-dereferences (even those in the heap). Both of these systems are considerably slower than ProPolice, capable of slowing down an application by even an order of magnitude, making them unsuitable for use by production systems. Fortunately, their impact on performance is less relevant in our approach.

3.2 TXL

Armed with the information produced by ProPolice, the code-transformation component of our system, TXL, is invoked. TXL is a hybrid functional and rule-based language which is well-suited for performing source-to-source transformation and for rapidly prototyping new languages and language processors. The grammar responsible for parsing the source input is specified in a notation similar to Extended Backus-Nauer (BNF). Several parsing strategies are supported by TXL making it comfortable with ambiguous grammars allowing for more "natural" user-oriented grammars, circumventing the need for strict compiler-style "implementation" grammars [68].

In our system, we use TXL for C-to-C transformations by making changes to the ANSI C grammar. In particular we move statically defined variables from the stack to the heap, using the TXL program shown in Appendix A. This is achieved by examining declarations in the source and transforming them to pointers where the size is allocated with a *malloc()* function call. Furthermore, we adjust the C grammar to free the variables before the function returns. After making changes to the standard ANSI C grammar that allow entries such as *malloc()* to be inserted between declarations and statements, the transformation step is trivial. The "number" and the "id" in this example refer to the size and name of the allocated buffer respectively, which are constructed by the *NewD()* TXL function. Shown in the example are also the parameters used to identify which buffer and function should be transformed by the TXLargs, which are the arguments passed to TXL. The other heuristic we use (not shown in Appendix A) is "slice-off" functionality. There, we use TXL to simply comment out the code of the superfluous function and embed a "return" in the function.

Figure 2. **Protected Malloc: Write-protected memory pages surround a buffer allocated with** *pmalloc().*

In the move-to-heap approach, we use an alternative *malloc()* implementation we developed specifically for this purpose. *pmalloc()* allocates two additional, zero-filled write-protected memory pages that surround the requested allocated memory region, as shown in Figure 2. Any buffer overflow or underflow will cause the operating system to issue a Segmentation Violation signal (SIGSEGV) to the process. We use *mprotect()* to mark the surrounding pages as read-only. This functionality is similar to that offered by the *ElectricFence* memory-debugging library.

Our TXL program inserts a *setjmp()* call immediately before the function call that caused the buffer overflow, as shown in Appendix C. The effect of this operation is to save the stack pointers, registers, and program counter, such that the program can later restore their state. We also inject a signal handler that catches the SIGSEGV and calls *longjmp()*, restoring the stack pointers and registers (including the program counter) to their values prior the call to the offending function (in fact, they are restored to their values as of the call to *setjmp()*). The program will then re-evaluate the injected conditional statement that includes the *setjmp()* call. This time, however, the return value will cause the conditional to evaluate to false, thereby skipping execution of the offending function. Note that the targeted buffer will contain exactly the amount of data (infection vector) it would if the offending function performed correct data-truncation.

There are two benefits in this approach. First, objects in the heap are protected from being overwritten by an attack on the specified variable, since there is a signal violation when data is written beyond the allocated space. Second, we can recover gracefully from an overflow attempt, since we can recover the stack context environment prior to

the offending function's call, and effectively *longjmp()* to the code immediately following the routine that caused the overflow or underflow.

Examination of the source code of the programs featured in the CoSAK study illustrated that the majority of the calls that caused an overflow/underflow (*e.g., strcpy(), memcpy(), etc.*) did not check for return values or include calls to other routines. This is an important observation since it validates our assumption that the heuristic can circumvent the malignant call using *longjmp()*.

3.3 Sandboxed Environment

Finally, for our sandboxed environment we use the VMWare virtual machine where we run the OpenBSD operating system [105]. VMWare allows operating systems and software applications to be isolated from the underlying operating system in secure virtual machines that co-exist on a single piece of hardware. Once we have created a correct version of the application, we simply update its image on the production environment outside the virtual environment, and restart it.

4 Experimental Evaluation

In order to illustrate the capabilities of our system and the effectiveness of the patch heuristics, we constructed a simple file-serving application that had a buffer overflow vulnerability and contained superfluous services where we could test against stack-smashing attacks and slice-of functionality respectively. For these purposes, the application used a simple two-phase protocol where a service is requested (different functions) and then the application waits for network input. The application was written in ANSI C.

A buffer overflow attack was constructed that overwrites the return address and attempts to get access to a root shell. The application was compiled under OpenBSD with the ProPolice extensions to GCC. Armed with the knowledge provided by ProPolice, the names of the function and buffer potentially responsible for the buffer overflow, the TXL implementation of our heuristics is invoked. Specific to the set of actions that we have implemented thus far, we test the heuristics and recompile the TXL-transformed code, and run a simple functionality test on the application (whether it can correctly serve a given file). The test is a simple script that attempts to access the available service. This application was an initial proof-of-concept for our system, and did not prove the correctness of our approach. More substantial results were acquired through the examination of the applications provided by the Code Security Analysis Kit project.

4.1 CoSAK data

In order to further test our heuristics, we examined a number of vulnerable open-source software products. This data was made available through the Code Security Analysis Kit (CoSAK) project from the software engineering research group at Drexel university. CoSAK is a DARPA-funded project that is developing a toolkit for software auditors to assist with the development of high-assurance and secure software systems. They have compiled a database of thirty OSS products along with their known vulnerabilities and respective patches. This database is comprised of general vulnerabilities, with a large number listed as susceptible to buffer overflow attacks. The move-to-heap heuristic was tested against this data set and the results are illustrated in Appendix B. Note that many of these applications are not in fact network services, and would thus probably not be susceptible to a worm. However, they should serve as a representative sample of buffer overflow vulnerabilities.

As illustrated in Appendix B, we tested the move-to-heap heuristic against the CoSAK data-set, which resulted in fixing 14 out of 17 "fixable" buffer overflow vulnerabilities, or 82% success rate. The remaining 14 products were not tested because their vulnerabilities were unrelated (non buffer-overflow). The products that were not amenable to the heuristic were examined, and in all cases what would be required to provide an appropriate fix would be adjustments to the TXL heuristics to cover special cases, such as handling multi-dimensional buffers and pre-initialized arrays.

The majority of the vulnerabilities provided by the CoSAK dataset were caused by calls to the *strcpy()* routine. Examination of the respective security patches showed that for most cases the buffer overflow susceptibility could be repaired by a respective *strncpy()*. Furthermore, most routines did not check for return values and did not include routines within the routines, thus providing fertile ground for use of our *pmalloc()* heuristic.

4.2 Performance

In order to evaluate the performance of our system, we tested the patch-generation engine on an instrumented version of Apache 2.0.45. Apache was chosen due to its popularity [5] and source-code availability. Basic Apache functionality was tested, omitting additional modules. The purpose of the evaluation was to validate the hypothesis that heuristics can be applied and tested in a time-efficient manner. The tests were conducted on a PC with an AMD Athlon processor operating at 800MHz and 512MB of RAM. The underlying operating system was OpenBSD 3.3.

One assumption that our system makes is that the instrumented application is already compiled in the sandboxed environment so that any patch heuristic would not require a complete re-compilation of the application. In order to get a realistic insight on the time that would be required from applying a patch and being able to test the application, we applied our move-to-heap TXL transformation on a number of different files, ranging from large to small sizes, and recompiled the latest version of Apache. The ranged average for compilation and relinking was 8.3 seconds.

Another important issue in terms of performance is the TXL transformation time for our basic heuristics. By being able to pass the specific function name and buffer to TXL, the transformation time is greatly reduced as the rule-set is concentrated on a targeted section of the source code. The average transformation time for different functions that were examined was 0.045 seconds. This result is very encouraging as it allows the assumption that the majority of the heuristics can be applied and tested in under 10 seconds.

5 Discussion

5.1 Challenges

There are several challenges associated with our approach:

1. **Determination of the nature of the attack (*e.g.,* buffer overflow), and identification of the likely software flaws that permit the exploit.** Obviously, our approach can only fix already-known attacks, *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 [36] to identify the likely functions and buffers that lead to the overflow condition. More powerful analysis tools [85, 58, 14, 89] can be easily employed in our architecture to catch more sophisticated code-injection attacks, and we intend to investigate them in future work. 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.

2. **Reliable repairing of the software.** 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 [68], 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. One interesting possibility is the use of Aspect-Oriented Programming to create locations ("hooks") in the source code that would allow the insertion of appropriate fixes. We plan to investigate this in future research.

   Interestingly, our architecture could be used to automatically fix any type of software fault, such as invalid memory dereferences, by plugging-in the appropriate repair module. When it is impossible to automatically obtain a software fix, we can use content-filtering as in [84] to temporarily protect the service. The possibility of combining the two techniques is a topic of future research.

3. **Source-code availability.** 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 [78] 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.

4. Finally, with respect to multi-partite worms, *i.e.,* worms using multiple independent infection vectors and propagation mechanisms (*e.g.,* spreading over both email and HTTP), our architecture treats such infections as independent worms.

5.2 Centralized *vs.* Distributed Reaction

The authors of [99] envision a Cyber "Center for Disease Control" (CCDC) for identifying outbreaks, rapidly analyzing pathogens, fighting the infection, and proactively devising methods for 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. Furthermore, fixes would still need to be handcrafted by humans and thus arrive too late to help in worm containment. 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.

When an exploitable vulnerability is discovered, our architecture could be used by the CCDC to distribute "fake worms". This channel would be treated as another sensor supporting the analysis engine. Propagation of these fake worms would trigger the creation of a quick-fix if the warning is deemed authentic (*i.e.,* the application crashes as a result of running the attack in the sandbox). Again, this would serve as a mechanism for distributing quick patches by independent parties, by distributing only the exploit and allowing organizations to create their own patches.

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. A similar approach is taken by some managed-security vendors, who keep a number of programmers available on a 24-hour basis. In all cases, administrators must be aware of the services offered (officially or unofficially) by all the hosts in their networks.

5.3 Attacks Against the System

Naturally, our system should not create new opportunities for attackers to subvert applications and hosts. One concern 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 [32] or IPsec [54].

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 [10]. 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.

As with any fully-automated task, the risks of relying on automated patching and testing as the only real-time verification techniques are not fully understood. To the extent that our system correctly determines that a buffer overflow attack is possible, the system's operation is safe: either a correct patch for the application will be created, or the application will have to be shut-down (or replaced with a non-working version). Considering the alternative, *i.e.,* guaranteed loss of service *and* subversion of the application, we believe that the risk will be acceptable to many. The question then centers around the correctness of the analysis engine. Fundamentally, this appears to be an impossible problem — our architecture enables us to add appropriate checks as needed, but we cannot guarantee absolute safety.

6 Related Work

Computer viruses are not a new phenomenon, and they have been studied extensively over the last several decades. Cohen was the first to define and describe computer viruses in their present form. In [26], he gave a theoretical basis for the spread of computer viruses. In [99], 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. In [98], the same authors show how a worm using pre-compiled lists of IP addresses known to be vulnerable can infect one million hosts in half a second. The strong analogy between biological and computer viruses [42] led Kephart *et al.* to investigate the propagation of computer viruses based on epidemiological models. In [55], they extend the standard epidemiological model by placing it on a directed graph, and use a combination of analysis and simulation to study its behavior. They conclude that if the rate at which defense mechanisms detect and remove viruses is sufficiently high, relative to the rate at which viruses spread, they are adequate for preventing widespread propagation of viruses.

Since the first Internet-wide worm [94], considerable effort has gone into preventing worms from exploiting common software vulnerabilities by using the compiler to inject run-time safety checks into applications [30, 50, 27, 36, 38, 104, 28, 16, 78, 86], safe languages and APIs [14, 48, 69], and static [23, 61, 23, 13, 24, 35, 22, 49] or dynamic [64, 62] analysis tools. While shortcomings may be attributed to each of these tools or approaches individually (*e.g.,* [20, 111]), the fact is that they have not seen wide use. We speculate that the most important reasons are: complexity; performance implications (or a perception of such); and, perhaps most importantly, a requirement for proactiveness on the part of application developers, who are often under pressure to meet deadlines or have no incentive to use new-fangled software verification tools.

Another approach has been that of containment of infected applications, exemplified by the "sandboxing" paradigm (*e.g.,* [43, 29, 58, 79, 75, 80, 88, 44, 45, 40]). Unfortunately, even when such systems are successful in containing the virus [39], they do not always succeed in preventing further propagation or ensuring continued service availability [65]. Furthermore, there is often a significant performance overhead associated with their use, which deters many users from taking advantage of them. User-level sandboxing approaches [81] using a second monitor process seem more promising, especially with the appearance of automated tools [18] and libraries [56] to assist the programmer; however, these require access and significant modifications to the source code of the application.

The Code-Red worm [6] was analyzed extensively in [118]. The authors of that work conclude that even though epidemic models can be used to study the behavior of Internet worms, they are not accurate enough because they cannot capture some specific properties of the environment these operate in: the effect of human countermeasures against worm spreading (*i.e.,* cleaning, patching, filtering, disconnecting, *etc.*), and the slowing down of the worm infection rate due to the worm's impact on Internet traffic and infrastructure. They derive a new general Internet worm model called *two-factor worm* model, which they then validate in simulations that match the observed Code Red data available to them. Their analysis seems to be supported by the data on Code Red propagation in [70] and [93] (the latter distinguished between different worms that were active simultaneously active). A similar analysis on the SQL "Slammer" (Sapphire) worm [7] can be found in [10], and for Witty in [90]. More recent analyses [117] show that it is possible to predict the overall vulnerable population size using Kalman filters early in the propagation cycle of a worm, allowing for detection of a fast-spreading worm when only 1% or 2% of vulnerable computers on the network have been infected.

Code-Red inspired several countermeasure technologies, such as La Brea [66], which 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 [112, 103], each host restricts the rate at which connections may be initiated. If adopted universally, such an approach would reduce the spreading rate of a worm by up to an order of magnitude, without affecting legitimate communications.

[114] detect worms by monitoring probes to unassigned IP addresses ("dark space") or inactive ports and computing statistics on scan traffic, such as the number of source/destination addresses and the volume of the captured traffic. By measuring the increase on the number of source addresses seen in a unit of time, it is possible to infer the existance of a new worm when as little as 4% of the vulnerable machines have been infected. A similar approach for isolating

infected nodes inside an enterprise network [97] is taken in [51], where it was shown that as little as 4 probes may be sufficient in detecting a new post-scanning worm. [109] describes an approximating algorithm for quickly detecting scanning activity that can be efficiently implemented in hardware. [87] describes a combination of reverse sequential hypothesis testing and credit-based connection throttling to quickly detect and quarantine local infected hosts. These systems are effective only against scanning worms (not topological, or "hit-list" worms), and rely on the assumption that a most scans will result in non-connections. As such, they as susceptible to false positives, either accidentally (*e.g.,* when a host is joining a peer-to-peer network such as Gnutella, or during a temporary network outage) or on purpose (*e.g.,* a malicious web page with many links to images in random/not-used IP addresses). Furthermore, it may be possible for several instances of a worm to collaborate in providing the illusion of several successful connections, or to use a list of *known repliers* to blind the anomaly detector.

[57] describes an algorithm for correlating packet payloads from different traffic flows, toward deriving a worm signature. The technique is promising, although further improvements are required to allow it to operate in real time. Earlybird [92] presents a more practical algorithm for doing payload sifting, and correlates these with a range of unique sources generating infections and destinations being targeted. However, polymorphic and metamorphic worms [100] remain a challenge; Spinelis [95] showed that reliably detecting viruses is an NP-hard problem. Buttercup [74] attempts to detect polymorphic buffer overflow attacks by identifying the ranges of the possible return memory addresses for existing buffer overflow vulnerabilities. Unfortunately, this heuristic cannot be employed against some of the more sophisticated overflow attack techniques [76]. Furthermore, the false positive rate is very high, ranging from 0.01% to 1.13%. Vigna *et al.* [106] discuss a method for testing detection signatures against mutations of known vulnerabilities to determine thei quality of the detection model and mechanism.

[71] describes a design space of worm containment systems using three parameters: reaction time, containment strategy, and deployment scenario. The authors use a combination of analytic modeling and simulation to describe how each of these design factors impacts the dynamics of a worm epidemic. Their analysis suggests that there are significant gaps in containment defense mechanisms that can be employed, and that considerable more research (and better coordination between ISPs and other entities) is needed. However, their analysis focuses exclusively on containment mechanisms (*i.e.,* network filtering), which they consider the only viable defense mechanism. In [108], the authors describe a mechanism for pushing to workstations vulnerability-specific, application-aware filters expressed as programs in a simple language. These programs roughly mirror the state of the protected service, allowing for more intelligent application of content filters, as opposed to simplistic payload string matching.

One approach for detecting new email viruses was described in [17], which keeps track of email attachments as they are sent between users through a set of collaborating email servers that forward a subset of their data to a central data warehouse and correlation server. Only attachments with a high frequency of appearance are deemed suspicious; furthermore, the email exchange patterns among users are used to create models of normal behavior. Deviation from such behavior (*e.g.,* a user sending a particular attachment to a large number of other users at the same site, to which she has never sent email before) raises an alarm. Naturally, an administrator has to examine the available data and determine whether the attachment really constitutes a virus, or is simply a very popular message. Information about dangerous attachments can be sent to the email servers, which then filter these out. One interesting result from this work is that their system only need be deployed to a small number of email servers, such that it can examine a minuscule amount of email traffic (relative to all email exchanged on the Internet) — they claim 0.1% — before they can determine virus outbreaks and be able to build good user behavior models. As similar technique, tracking attachments through the network, is described by Xiong [115]. An attempt to apply behavior-based detection at the network layer for worm detection is described in [34].

Wong *et al.* [113] study the behavior of the SoBig and MyDoom mass-mailing worms using network packet traces from the CMU network. They identify DNS servers as a possible location for slowing down mass-mailing worms. In constrast, monitoring outgoing mail on SMTP servers is unlikely to work, since most such worms contain their own SMTP engines.

[102] proposes the use of "predator" viruses, which are effectively good-will viruses that spread in much the same way malicious viruses do but try to eliminate their designated "victim" viruses. The authors model the interaction between predators and other viruses by equations used in mathematical biology, and show that predators can be made to perform their tasks without flooding the network and consuming all available resources. In practice, however, viruses would be likely to patch the vulnerabilities they exploited (as recent worms in fact do, after an Code Red

anti-worm was released soon after Code Red itself was released on the Internet). Thus, designers of predators would have to find their own exploits (or safeguard exploits for future use), which is not an attractive proposition. One encouraging result of their work was that the number of initial predators needed to contain a highly-aggressive virus could be as small as 2,000. Castaneda *et al.* [21] describe a system for automatically creating anti-worms, and analyze (via simulation) its effectiveness against CodeRed, Blaster and Slammer.

The HACQIT architecture [52, 84, 82, 83] 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. [53] contains a general discussion on the use of diversity for preventing monoculture-based attacks. [101] 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) [4] 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. Porras *et al.* [77] argue that hybrid defenses using complementary techniques (in their case, connection throttling at the domain gateway and a peer-based coordination mechanism), can be much more effective against a wide variety of worms.

Nojiri *et al.* [73] present a cooperative response algorithm where edge-routers share attack reports a small set of other edge-routers. Edge routers update their alert level based on the shared attack reports and decide whether to enable traffic filtering and blocking for a particular attack. Indra [47] also takes a cooperative approach to the problem of worm detection. It uses a peer-to-peer approach for exchanging infection information with trusted nodes. They define as "trusted" those nodes for which they can discover a public key and a binding to an IP address. However, Indra does not address the problem of subverted nodes that spread false information. [59] takes a unique approach to intrusion detection, which can be used to cover worm propagation. They specify an algorithm for determining sampling rates along the min-cut of a network graph to maximize the detection of malicious packets, while minimizing the expended resources. However, this approach may be impractical because most organization's networks more closely resemble trees (not graphs) with well-defined traffic-ingress points.

[107] presents some very encouraging results for slowing down the spread of viruses. The authors simulated the propagation of virus infections through certain types of networks, coupled with partial immunization. Their findings show that even with low levels of immunization, the infection slows down significantly. Their experiments, however, looked at a single virus. Our work investigates the detection of potentially multiple viruses when there is no *a priori* knowledge of which viruses may attack. We use a distributed set of nodes that search for viruses in the data flowing though the network and arriving at end-nodes. We aim to maximize the probability that any individual virus (eventually) encounters a level of immunization that will retard its growth.

In the realm of "traditional" computer viruses, 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, *e.g.,* using partial simulation during program execution. This has led to co-evolution [72], an ever-escalating arms race between virus writers and anti-virus developers, which the anti-virus developers are not likely to win in the long run [95].

Lin, Ricciardi, and Marzullo study how computer worms affect the availability of services. In [65], they study the fault tolerance of multicast protocols under self-propagating virus attacks.

Leavitt [63] discusses the threat of worms aimed at mobile phones, describing some of the first malware of this type. Zhou *et al.* [116] discuss worms that spread over peer-to-peer networks, exploiting the richer (and arbitrary) topologies to achieve accurate targetting and fast propagation.

7 Conclusion

We argued that increased use of end-to-end encryption and worm stealthiness, as well as the inadequacy of existing preventive mechanisms to ensure service availability in the presence of software flaws, necessitate the development of an end-point worm-reaction approach that employs invasive but targeted mechanisms to fix the vulnerabilities. We presented an architecture for countering 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 counter specific buffer-overflow instances. If we manage to create a version of the application that is both resistant to the worm and meets certain minimal-functionality criteria, embodied in a functionality test-suite created in advance by the system administrator, we update the production servers.

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 [99]. Comprehensive security measures can be administered at a later time. Furthermore, our architecture is easily extensible to accommodate detection and reactive measures against new types of attacks as they become known. Our experimental analysis, using a number of known vulnerable applications are hypothetical targets of a worm infection, shows that our architecture can fix 82% of all such attacks, and that the maximum time to repair a complicated application was less than 8.5 seconds. We believe that these preliminary results validate our approach and will spur further research.

References

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

[2] OC48 Analysis – Trace Data Stratified by Applications. http://www.caida.org/analysis/workload/byapplication/oc48/port_analysis_app.xml.

[3] The Code Security Analysis Kit (CoSAK). `http://serg.cs.drexel.edu/cosak/index.shtml/`.

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

[5] Web Server Survey. `http://www.securityspace.com/s_survey/data/200304/`.

[6] 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.

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

[8] CERT Advisory CA-2003-19: Exploitation of Vulnerabilities in Microsoft RPC Interface. `http://www.cert.org/advisories/CA-2003-19.html`, July 2003.

[9] CERT Advisory CA-2003-21: W32/Blaster Worm. `http://www.cert.org/advisories/CA-2003-20.html`, August 2003.

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

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

[12] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online Patches and Updates for Security. In *Proceedings of the* 14^{th} *USENIX Security Symposium*, pages 287–302, August 2005.

[13] K. Ashcraft and D. Engler. Detecting Lots of Security Holes Using System-Specific Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2002.

[14] 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.

[15] S. M. Bellovin. Distributed Firewalls. *;login: magazine, special issue on security*, November 1999.

[16] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, August 2003.

[17] M. Bhattacharyya, M. G. Schultz, E. Eskin, S. Hershkop, and S. J. Stolfo. MET: An Experimental System for Malicious Email Tracking. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 1–12, September 2002.

[18] D. Brumley and D. Song. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 57–71, August 2004.

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

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

[21] F. Castaneda, E. C. Sezer, and J. Xu. WORM vs. WORM: Preliminary Study of an Active Counter-Attack Mechanism. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 83–93, October 2004.

[22] H. Chen, D. Dean, and D. Wagner. Model Cehcking One Million Lines of C Code. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 171–185, February 2004.

[23] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 235–244, November 2002.

[24] B. Chess. Improving Computer Security Using Extended Static Checking. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2002.

[25] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th USENIX Security Symposium*, pages 169–186, August 2003.

[26] F. Cohen. Computer Viruses: Theory and Practice. *Computers & Security*, 6:22–35, February 1987.

[27] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. Formatguard: Automatic protection from printf format string vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Aug. 2001.

[28] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, August 2003.

[29] C. Cowan, S. Beattie, C. Pu, P. Wagle, and V. Gligor. SubDomain: Parsimonious Security for Server Appliances. In *Proceedings of the 14th USENIX System Administration Conference (LISA 2000)*, March 2000.

[30] 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* 7^{th} *USENIX Security Symposium*, January 1998.

[31] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. HoneyStat: Local Worm Detection Using Honepots. In *Proceedings of the* 7^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 39–58, October 2004.

[32] T. Dierks and C. Allen. The TLS protocol version 1.0. RFC 2246, IETF, Jan. 1999.

[33] 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.

[34] D. R. Ellis, J. G. Aiken, K. S. Attwood, and S. D. Tenaglia. A Behavioral Approach to Worm Detection. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 43–53, October 2004.

[35] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of ACM SOSP*, October 2003.

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

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

[38] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th USENIX Security Symposium*, pages 55–66, August 2001.

[39] 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.

[40] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 187–201, February 2004.

[41] 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.

[42] S. Goel and S. F. Bush. Biological Models of Security for Virus Propagation in Computer Networks. *USENIX ;login:*, 29(6):49–56, December 2004.

[43] 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.

[44] S. Ioannidis, S. Bellovin, and J. M. Smith. Sub-Operating Systems: A New Approach to Application Security. In *Proceedings 10th SIGOPS European Workshop*, pages 108–115, September 2002.

[45] S. Ioannidis and S. M. Bellovin. Building a Secure Browser. In *Proceedings of the Annual USENIX Technical Conference, Freenix Track*, pages 127–134, June 2001.

[46] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a Distributed Firewall. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 190–199, November 2000.

[47] R. Janakiraman, M. Waldvogel, and Q. Zhang. Indra: A peer-topeer approach to network intrusion detection and prevention. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Workshop on Enterprise Security*, June 2003.

[48] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX Annual Technical Conference*, pages 275–288, June 2002.

[49] R. Johnson and D. Wagner. Finding User/Kernel Pointer Bugs With Type Inference. In *Proceedings for the* 13^{th} *USENIX Security Symposium*, pages 119–134, August 2004.

[50] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*, 1997.

[51] J. Jung, V. Paxson, A. W. Berger, and H. Balakrishnan. Fast Portscan Detection Using Sequential Hypothesis Testing. In *Proceedins fo the IEEE Symposium on Security and Privacy*, May 2004.

[52] J. E. Just, L. A. Clough, M. Danforth, K. N. Levitt, R. Maglich, J. C. Reynolds, and J. Rowe. Learning Unknown Attacks – A Start. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID)*, October 2002.

[53] J. E. Just and M. Cornwell. Review and Analysis of Synthetic Diversity for Breaking Monocultures. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 23–32, October 2004.

[54] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, IETF, Nov. 1998.

[55] J. O. Kephart. A Biologically Inspired Immune System for Computers. In *Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*, pages 130–139. MIT Press, 1994.

[56] D. Kilpatrick. Privman: A library for partitioning applications. In *Proceedings of the USENIX Technical Conference, Freenix Track*, pages 273–284, June 2003.

[57] H. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 271–286, August 2004.

[58] 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.

[59] M. Kodialam and T. V. Lakshman. Detecting Network Intrusions via Sampling: A Game Theoretic Approach. In *Proceedings of the 22nd Annual Joint Conference of IEEE Computer and Communication Societies (INFOCOM)*, April 2003.

[60] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 255–270, August 2004.

[61] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, pages 177–190, August 2001.

[62] E. Larson and T. Austin. High Coverage Detection of Input-Related Security Faults. In *Proceedings of the 12th USENIX Security Symposium*, pages 121–136, August 2003.

[63] N. Leavitt. Mobile Phones: The Next Frontier for Hackers? *IEEE Computer*, 38(4), April 2005.

[64] 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.

[65] 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.

[66] T. Liston. Welcome To My Tarpit: The Tactical and Strategic Use of LaBrea. `http://www.threenorth.com/LaBrea/LaBrea.txt`, 2001.

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

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

[69] T. C. Miller and T. de Raadt. strlcpy and strlcat: Consistent, Safe, String Copy and Concatentation. In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, June 1999.

[70] 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.

[71] 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.

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

[73] D. Nojiri, J. Rowe, and K. Levitt. Cooperative Response Strategies for Large Scale Attack Mitigation. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX)*, pages 293–302, April 2003.

[74] A. Pasupulati, J. Coit, K. Levitt, S. Wu, S. Li, J. Kuo, and K. Fan. Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities. In *Proceedings of the Network Operations and Management Symposium (NOMS)*, pages 235–248, vol. 1, April 2004.

[75] D. S. Peterson, M. Bishop, and R. Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, August 2002.

[76] J. Pincus and B. Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overflows. *IEEE Security & Privacy*, 2(4):20–27, July/August 2004.

[77] P. Porras, L. Briesemeister, K. Levitt, J. Rowe, and Y.-C. A. Ting. A Hybrid Quarantine Defense. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 73–82, October 2004.

[78] M. Prasad and T. Chiueh. A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224, June 2003.

[79] V. Prevelakis and D. Spinellis. Sandboxing Applications. In *Proceedings of the USENIX Technical Annual Conference, Freenix Track*, pages 119–126, June 2001.
[80] N. Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, August 2003.
[81] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 231–242, August 2003.
[82] J. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich. On-line Intrusion Protection by Detecting Attacks with Diversity. In *16th Annual IFIP 11.3 Working Conference on Data and Application Security Conference*, April July.
[83] J. C. Reynolds, J. Just, L. Clough, and R. Maglich. On-Line Intrusion Detection and Attack Prevention Using Diversity, Generate-and-Test, and Generalization. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS)*, January 2003.
[84] J. C. 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.
[85] 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.
[86] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 159–169, February 2004.
[87] S. E. Schechter, J. Jung, and A. W. Berger. Fast Detection of Scanning Worm Infections. In *Proceedings of the* 7^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 59–81, October 2004.
[88] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications. In *Proceedings of ACM SOSP*, October 2003.
[89] J. Seward and N. Nethercote. Valgrind, an open-source memory debugger for x86-linux. `http://developer.kde.org/~sewardj/`.
[90] C. Shannon and D. Moore. The Spread of the Witty Worm. *IEEE Security & Privacy*, 2(4):46–50, July/August 2004.
[91] J. F. Shoch and J. A. Hupp. The "worm" programs – early experiments with a distributed computation. *Communications of the ACM*, 22(3):172–180, March 1982.
[92] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated worm fingerprinting. In *Proceedings of the* 6^{th} *Symposium on Operating Systems Design & Implementation (OSDI)*, December 2004.
[93] D. Song, R. Malan, and R. Stone. A Snapshot of Global Internet Worm Activity. Technical report, Arbor Networks, November 2001.
[94] E. H. Spafford. The Internet Worm Program: An Analysis. Technical Report CSD-TR-823, Purdue University, 1988.
[95] D. Spinellis. Reliable identification of bounded-length viruses is NP-complete. *IEEE Transactions on Information Theory*, 49(1):280–284, January 2003.
[96] L. Spitzner. *Honeypots: Tracking Hackers*. Addison-Wesley, 2003.
[97] S. Staniford. Containment of Scanning Worms in Enterprise Networks. *Journal of Computer Security*, 2004. (to appear).
[98] S. Staniford, D. Moore, V. Paxson, and N. Weaver. The Top Speed of Flash Worms. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 33–42, October 2004.
[99] 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.
[100] P. Ször and P. Ferrie. Hunting for Metamorphic. Technical report, Symantec Corporation, June 2003.
[101] T. Toth and C. Kruegel. Connection-history Based Anomaly Detection. In *Proceedings of the IEEE Workshop on Information Assurance and Security*, June 2002.
[102] H. Toyoizumi and A. Kara. Predators: Good Will Mobile Codes Combat against Computer Viruses. In *Proceedings of the New Security Paradigms Workshop (NSPW)*, pages 13–21, September 2002.
[103] J. Twycross and M. M. Williamson. Implementing and testing a virus throttle. In *Proceedings of the 12th USENIX Security Symposium*, pages 285–294, August 2003.
[104] Vendicator. Stack shield. `http://www.angelfire.com/sk/stackshield/`.
[105] G. Venkitachalam and B.-H. Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor.
[106] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits, October 2004.
[107] 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.
[108] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM Conference*, pages 193–204, August 2004.
[109] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, pages 29–44, August 2004.
[110] 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.

[111] 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.

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

[113] C. Wong, S. Bielski, J. M. McCune, and C. Wang. A Study of Mass-Mailing Worms. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 1–10, October 2004.

[114] J. Wu, S. Vangala, L. Gao, and K. Kwiat. An Effective Architecture and Algorithm for Detecting Worms with Various Scan Techniques. In *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pages 143–156, February 2004.

[115] J. Xiong. ACT: Attachment Chain Tracing Scheme for Email Virus Detection and Control. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, pages 11–22, October 2004.

[116] L. Zhou, L. Zhang, F. M. Sherry, N. Immorlica, M. Costa, and S. Chien. A First Look at Peer-to-Peer Worms: Threats and Defenses. In *Proceedings of the* 4^{th} *International Workshop on Peer-To-Peer Systems (IPTPT)*, February 2005.

[117] C. C. Zou, L. Gao, W. Gong, and D. Towsley. Monitoring and Early Warning for Internet Worms. In *Proceedings of the 10th ACM International Conference on Computer and Communications Security (CCS)*, pages 190–199, October 2003.

[118] 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.

Appendix A. TXL Code

```
......
rule argsMalloc
replace [function_definition]
  DECL_SPECIFIERS [opt decl_specifiers]
  DECLARATOR_ [declarator]
  OPTIONAL_DECL [opt NL_declarations]
  '{ D [declarations] S [statements] '}

import TXLargs [repeat stringlit]
deconstruct * TXLargs
  "-myoption" FunctionName [stringlit]
  buffer [stringlit]
export Buf [stringlit]
  buffer
deconstruct * DECLARATOR_
  R[repeat ptr_operator]
  I[id]
  T[repeat declarator_extension]
construct NewD [declarations]
  D [test]
import Number [number]
import Id [id]
where
  I [= 'FunctionName]

by
  DECL_SPECIFIERS
  DECLARATOR_
  OPTIONAL_DECL
    '{ NewD Id '=pmalloc( Number '); S 'pfree( Id '); '}
end rule
......
```

Appendix B. CoSAK Data

The column ”Functions within functions” indicates whether the vulnerable system call used in the application invoked another function as part of the parameters to the call. The column ”Return value” indicates whether the vulnerable system call's return value was checked upon returning from the call. The significance of these columns is pertinent to the application of our *pmalloc()* heuristic.

Appendix C. Oveflow Recovery Code

```
#include <setjmp.h>              /* ADDED */
#include <signal.h>              /* ADDED */
jmp_buf worm_env; /* ADDED */

........

invoking_function ()
{
....
signal (SIGSEGV, worm_handler); /* ADDED */
....

if (setjmp (worm_env) == 0) {   /* ADDED */
        offending_function(...);
}                               /* ADDED */
....
}

int worm_handler () /* ADDED */
{                                /* ADDED */
    longjmp (worm_env, 1);       /* ADDED */
}                                /* ADDED */

.......
```