The Evolution of System-call Monitoring

Stephanie Forrest
Dept. of Computer Science
University of New Mexico
Albuquerque, NM USA
forrest@cs.unm.edu

Steven Hofmeyr
Lawrence Berkeley Laboratory
Berkeley, CA USA
shofmeyr@lbl.gov

Anil Somayaji
School of Computer Science
Carleton University
Ottawa, ON Canada
soma@scs.carleton.ca

Abstract

Computer security systems protect computers and networks from unauthorized use by external agents and insiders. The similarities between computer security and the problem of protecting a body against damage from externally and internally generated threats are compelling and were recognized as early as 1972 when the term computer virus was coined. The connection to immunology was made explicit in the mid 1990s, leading to a variety of prototypes, commercial products, attacks, and analyses. The paper reviews one thread of this active research area, focusing on system-call monitoring and its application to anomaly intrusion detection and response.

The paper discusses the biological principles illustrated by the method, followed by a brief review of how system call monitoring was used in anomaly intrusion detection and the results that were obtained. Proposed attacks against the method are discussed, along with several important branches of research that have arisen since the original papers were published. These include other data modeling methods, extensions to the original system call method, and rate limiting responses. Finally, the significance of this body of work and areas of possible future investigation are outlined in the conclusion.

1 Introduction

During the 1990's the Internet as we know it today grew from a small network of trusted insiders to a worldwide conglomerate of private citizens, governmental agencies, commercial enterprises, and academic institutions. As society at large embraced the Internet, opportunities and incentives for malicious activities exploded, creating demand for new computer security methods that could succeed in this open and uncontrolled environment. Open applications, mobile code and other developments helped erode the notion of a clear perimeter, which formerly separated a trusted system from its external environment. Previous approaches to computer security had emphasized top-down policy specification, provably correct implementations of policy, and deployment in correctly configured systems. Each of these assumptions became increasingly untenable, as the Internet grew and was integrated into human society.

The similarities between computer security in the age of the Internet and the problem of protecting a body against damage from internally and externally generated threats are compelling. They were recognized as early as 1972 when the term computer virus was introduced [67]. Later, Spafford argued that computer viruses are a form of artificial life [66], and several authors investigated the analogy between epidemiology and the spread of computer viruses across networks [38, 51, 59, 55]. The connection to immunology was made explicit in [15, 37], and since that time the ideas have been extended to incorporate significant amounts of immunology and to tackle ambitious computer security problems, including computer virus detection [15, 37], network security [24, 79, 33], spam filtering [57], and computer forensics [46].

As the primary defense system of the body, immune systems are a natural place to look for ideas about architectures and mechanisms for coping with dynamic threat environments. Immune systems detect foreign pathogens and misbehaving internal components (cells), and they choose and manage an effective response autonomously. Thus, the immune system can be thought of as a highly sophisticated intrusion detection and response system (IDS).

Despite debate in the immunological literature about how the immune system recognizes threats, in most of the work on computer immune systems it is assumed that natural immune systems work by distinguishing between protein fragments (peptides) that belong to the properly functioning body (*self*) and ones that come from invading and malfunctioning cells (*nonself*). To explore IDS designs that mimic those of the immune system, we must first decide what data or activity patterns will be used to distinguish between computational self and nonself. That is, we must decide what data streams the IDS will monitor and identify a threat model. To build a computer immune system, then, a computational analog to peptides must be found, one that will allow security violations to be detected without generating too many false alarms in response to routine changes in system behavior.

In this paper we focus our attention on the definition of self introduced in [14, 25, 65] to protect executing programs. In this work, discrimination between normal and abnormal behavior was based on what system calls (operating system services) are normally invoked by a running program. As a program executes, it might make several million system calls in a short period of time, and this signature of normal behavior is sufficient to distinguish between normal behavior and many attacks. Many attacks cause a vulnerable program to execute infrequently used code paths, which in turn leads to anomalous patterns of system calls. Anomaly detection based on system calls is able to detect intrusions that target a single computer, such as buffer overflow attacks, SYN floods, configuration errors, race conditions, and Trojan horses under the assumption that such attacks produce code execution paths that are not executed in normal circumstances.

A large number of researchers adopted the system-call approach, some seeking to improve on the original methods [49, 54, 45, 62, 29], some applying its method to other problems [68, 30, 50], and some attempting to defeat the system [74, 70]. Sana Security developed a product known as *Primary Response* based on the technology, which it actively marketed to protect servers. At this writing, the system-call method is the most mature application of biological principles to computer security. In this paper, we first review the general principles that guided our design decisions (Section 2); we next describe briefly the original system-call method and summarize the results it achieved (Section 3). Next, we summarize several important lines of research that have arisen since the original paper was published (Sections 4, 5, 6, 7). Finally Section 8 speculates on the significance of this body of work.

2 General Principles

The biological analogy led to a set of general design principles, which remain relevant to computer security more than a decade later. These include:

- A *generic mechanism* that provides coverage of a broad range of attacks, even if it is not 100% provably secure for any particular threat model. This approach is similar in spirit to "universal weak methods" in artificial intelligence that are applicable to many problems and do not require specialized domain knowledge. Although some of the pattern recognition mechanisms of the adaptive and innate immune systems are biased towards detecting important classes of infection, many are generic "danger detectors." The existence of autoimmune disease provides evidence that they are imperfect; yet they are highly effective at eliminating infections, and natural selection has conserved and enhanced immune systems in all higher animals. Choosing a ubiquitous and fundamental data stream, such as system calls, allowed us to design a system that could protect a wide variety of programs against a wide variety of threats without specialized knowledge.

- *Adaptable* to changes in the protected system. Computer systems, environments, and users are highly dynamic, and normal legitimate uses of systems are continually evolving. Just as biological defense systems need to cope with natural adaptive processes, so must computer security systems if they are to be robust. In our system, adaptability was achieved through the use of simple learning mechanisms, which were used to construct models of normal behavior and to update them over time.

- *Autonomy*: Computer systems have traditionally been thought of as tools that depend upon humans for guidance. However, as computer systems have become powerful, numerous, and interconnected, it is no longer feasible for humans to manage them directly. Biological systems necessarily operate independently, on-line, and in real-time because they live and interact with physical environments. In our system, we addressed this requirement with the most lightweight simple design we could think of—ignoring system call arguments, modeling data with simple data structures and without calculating probabilities or frequencies, and a generic response mechanism that slows down suspicous processes.

- *Graduated Response*: In computer security, responses tend to be binary, as in the case of access controls (either a user is allowed access or not), firewalls (either a connection is blocked or it is not), or cryptography (where either a file is encrypted or it is not). Biological systems have graduated responses, where small perturbations result in small responses, and large deviations cause more aggressive responses. We adopted this principle in process Homeostasis (pH), where system calls were delayed according to how many anomalous system calls had preceded them. Graduated responses allowed us to move away from the concept of security as a binary property and to tolerate imperfect detection.

- *Diversity* of the protection system. Natural immune systems are diverse. Individual differences arise both through genetic variations (e.g., MHC genes) or through life histories (e.g., which immune memories an individual has). Individual diversity promotes robustness, because an attack that escapes detection in one host is likely to be detected in another. Assuming that hosts have finite resources to devote to defense, this allows a higher level of population-level robustness than could be achieved by simply replicating the same defense system across systems. Beyond that, diversity of the protection system makes it much more difficult for an attacker to design a generic attack that will succeed everywhere. In our system, different environments and usage patterns confer diversity on each program invocation, which leads to diverse patterns of system calls observed during "normal" behavior. The extent of this diversity varies with the program [64, 25].

3 A Sense of Self for Unix Processes

Computer systems are vulnerable to external attack via many different routes. One of the most important of these is server programs that run with enhanced privilege, such as remote login servers (e.g., ssh), mail servers (e.g. sendmail), web-servers, etc. Software or configuration flaws in these running programs are exploited by attackers to gain illegitimate entry into systems, where they can take advantage of the privilege of the compromised process to seize control of the computer system. Privileged server processes form the main gateways for remote entry into a system, and hence it is vital to protect them from attack. These gateway server programs are frequently patched, but new vulnerabilities continue to be discovered, leading to widespread abuses such as the code-red worm [8], and many others [56].

One common approach is to scan the inputs to the server, usually at the network level. Such network intrusion detection systems typically scan for signatures of attacks in network packets. However, such systems are vulnerable to denial-of-service attacks, spoofing and other attacks [61], and they can only detect attacks for which they have signatures ahead of time. We believed that instead of focusing on inputs to servers, it would be better to monitor the runtime behavior of programs, because only code that is running can actually cause damage. And, it is harder to forge behavior than to forge data.

3.1 Behavioral characteristics

The central hypothesis of the original research [14] was that anomaly intrusion detection can provide an effective additional layer of protection for vulnerable privileged processes. An anomaly detection system develops a profile of normal behavior and monitors for deviations that indicate attacks. Traditionally, anomaly detection systems focused on characterizing user-behavior [3, 9, 44] and were often criticized for having high false positive rates because user behavior is erratic. The key to effective anomaly detection is to monitor a characteristic that is stable under normal, legitimate behavior, and perturbed by attacks.

Many characteristics could potentially be used in an anomaly detection system. Our choice was based on the observation that server programs tend to execute a limited set of tasks repeatedly, often with little variation. Those tasks correspond to regular paths through the program code. We chose a proxy in the form of short sequences of system calls executed by running programs. Our focus on system calls followed earlier work by Fink, Levitt and Ko [13, 39] that used system calls in a specification-based intrusion detection system. System call sequences can be monitored from the operating system, without necessitating recompilation or instrumentation of binary or source code, making the system extremely portable and potentially applicable to any program that exhibits regular code paths.

We wanted a system that was sufficiently lightweight that it could monitor all running programs in real-time, and even respond to prevent attacks before they caused harm (as was demonstrated in pH; see section 7). For this reason, we chose to define the normal profile using short sequences of system calls. There are many possible ways to represent short sequences of system calls, for example, lookahead pairs, n-grams, trees, etc. Figure 1 shows some of these representations, and they are described in Section 5. Most of our early work used lookahead pairs, although we experimented with exact sequences and Markov models. We disregarded system call parameters (the flow of data), to further simplify the problem. Our goal was to start as simply as possible, and later expand the system if required: the need for speed and real-time monitoring/response dictated that we discover the simplest possible mechanisms that would actually work.

3.2 Developing a normal profile

The first step in anomaly detection is gathering the data that will constitute the profile of normal behavior. We used two different methods for this: generating a *synthetic* normal profile by exercising a program in all of its anticipated normal modes of usage; and collecting *real* normal profiles by recording system call traces during normal, online usage of a production system[1]. The learning algorithm was used to determine the minimum set of short sequences of system calls that adequately defined normal: it's goal was to include all normally used code paths, but exclude those that are never used, even if those paths exist within the program

[1] An alternative approach is to determine normal through static analysis of the program code. See section 6.3

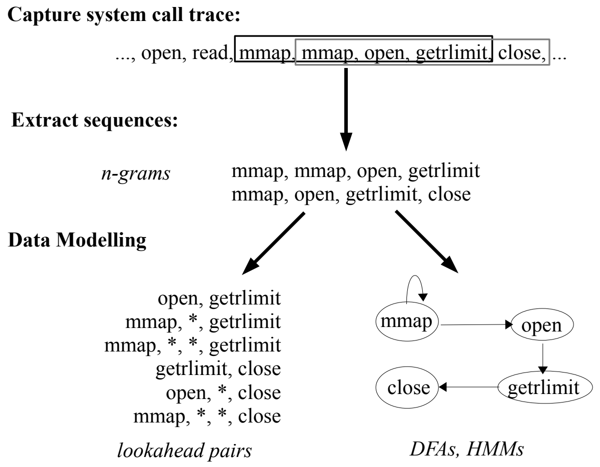

Figure 1. Representing system call streams

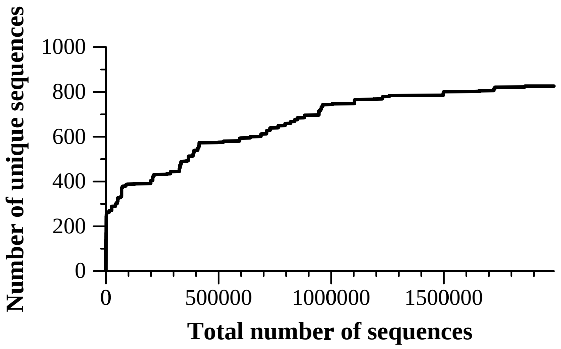

Figure 2. Growth of normal database for `lpr` from MIT's Artificial Intelligence Laboratory (reprinted from [76]).

source code. An important class of attacks involves injecting foreign code (e.g. buffer overflows), but many other attacks force a program to exercise existing but rarely used code paths, and hence do not require foreign code injection. Developing a normal profile is a typical machine learning problem: Undergeneralization leads to false positives, and overgeneralization leads to false negatives.

Experimentation with a wide range of programs (e.g. `sendmail`, `lpr`, `inetd`, `ftp`, `named`, `xlock`, `login`, `ssh`) demonstrated that these programs exhibit regular behavior and can be characterized by compact normal profiles [40, 25, 14]. Figure 2 shows how normal behavior converges to a fixed set of system call sequences of length 6 for `lpr` in a production environment—the Artificial Intelligence Laboratory at the Massachusetts Institute of Technology (MIT). The figure shows how initially there are many new sequences, but after a while few novel sequences are observed and the profile converges to a fixed size.

Not only can normal behavior be defined by limited sets of sequences of system calls, but what constitutes normal differs widely from one program to the next. For example, a typical run of `ftp` differed by between 28 and 35% (depending on the sequence length) from `sendmail` [14]. More importantly, different environments and usage patterns resulted in dramatically different normal, even for the identical program and operating system. For instance, the normal profile for `lpr` gathered at MIT differed markedly from the normal profile gathered at the University of New Mexico's Computer Science Department (UNM): only 29% of the unique sequences in the MIT profile were also present in the UNM profile. Later work on pH corroborated these results in lookahead pairs, showing 1) that two program profiles were 20-25% similar on average (over hundreds of programs) and 2) the same programs running on three different hosts with the same OS version differed, on average, in 22-25% of their lookahead pairs [64].

These results offered clear support for what we have termed the "Diversity Hypothesis": Normal code paths executed by a running program are highly dependent on typical usage patterns, configuration and environment, and hence can differ widely from one installation to the next, even for the same program and operating system. Diversity in biology confers robustness on a population, and can do the same for computer systems: an attack that succeeds against one implementation could fail against another because normal is different. Further, the attacker will not necessarily know a priori what constitutes normal for a given implementation—knowledge of the source code is not sufficient; also required is knowledge of the environment and usage patterns.

3.3 Detecting attacks

The normal profile must not only be stable and compact, but it must differ from behavior generated by attacks. Extensive experimentation demonstrated that normal sequences of system calls differ from a wide variety of attacks, including buffer overflows, SYN floods, configuration errors, race conditions, etc [14, 25, 40, 64]. This variety shows that the method is capable not only of detecting foreign code injection attacks (such as buffer overflows) but attacks that exercise unused code paths that exist in the program source. In contrast with methods based on static analysis, this last point illustrates the importance of a precise definition of normal that excludes unused source code paths.

One question of interest was determining the minimum necessary system call length required to detect all attacks.

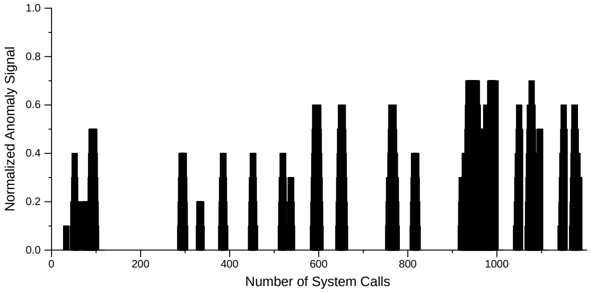

Figure 3. Anomaly signal of the `syslogd` **intrusion. (Reprinted from [25].)**

Shorter sequences are good because they result in more compact normal profiles, faster learning times, and less overhead during sequence matching. However, short sequences can potentially be more easily evaded (see section 4) and give less clear signals of an attack. A good minimal length was found to be 6 system calls [25], although for implementation reasons our later work used 8. Even at such short sequence lengths, attacks were usually obvious because they consisted of temporally clumped anomalies, as shown in Figure 3. In this figure, the x-axis represents time (in units of system calls) and the y-axis is the normalized number of recently seen anomalous system calls.

Many of the detected anomalies were not malicious. For example, the method detected errors such as forwarding loops, failed print jobs, and system misconfigurations. This illustrated one additional benefit of anomaly detection: It can be used to detect faults caused by non-malicious errors. This contrasts with a signature detection system, which usually has only the ability to detect known malicious events.

4 Subverting system call monitoring

Because most security properties of current software systems and algorithms cannot be proven, advances in security have long relied upon researchers studying systems and finding security vulnerabilities in them. *Mimicry* attacks were the earliest attempt to defeat the system call modeling approach. Wagner and Dean proposed that it was possible to craft sequences of system calls that exploited an attack, but appeared normal [74]. The trick to achieving this involves inserting "nullified" system calls, i.e. calls that have no effect, either because the return values are ignored or the parameters are manipulated. This enables an attacker to construct an attack sequence within a legitimate sequence using the "nullified" calls as padding. The mimicry has to persist as long as the attacker is exploiting the process that is being monitored, even once the initial penetration has succeeded. See Figure 4.

read() write() close() munmap() sigprocmask() wait4()
sigprocmask() sigaction() alarm() time() stat() read()
alarm() sigprocmask() *setreuid()* fstat() getpid()
time() write() time() getpid() sigaction() socketcall()
sigaction() close() flock() getpid() lseek() read()
kill() lseek() flock() sigaction() alarm() time()
stat() write() open() fstat() mmap() read() open()
fstat() mmap() read() close() munmap() brk() fcntl()
setregid() open() fcntl() *chroot()* *chdir()* setreuid()
lstat() lstat() lstat() lstat() open() fcntl() fstat()
lseek() getdents() fcntl() fstat() lseek() getdents()
close() write() time() open() fstat() mmap() read()
close() munmap() brk() fcntl() setregid() open() fcntl()
chroot() chdir() setreuid() lstat() lstat() lstat()
lstat() *open()* fcntl() brk() fstat() lseek() getdents()
lseek() getdents() time() stat() *write()* time() open()
getpid() sigaction() socketcall() sigaction() umask()
sigaction() alarm() time() stat() read() alarm()
getrlimit() pipe() fork() fcntl() fstat() mmap() lseek()
close() brk() time() getpid() sigaction() socketcall()
sigaction() chdir() sigaction() sigaction() write()
munmap() munmap() munmap() *exit()*

Figure 4. Sequence of system calls in a mimicry attack against `wuftpd`**. The underlined calls are part of the attack, all the rest are nullified calls. [75]**

However, there are limitations to such "nullified" mimicry attacks. First, the attacker needs to be able to *inject* the code containing the specially crafted sequence, which limits these mimicry attacks to only those that can exploit code-injection attacks, such as buffer overflows. Second, the diversity of normal profiles on different systems is a barrier because the attacker requires precise knowledge of the normal profile [75]. To what degree this defeats mimicry in practice has not been thoroughly investigated, but it is worth recalling that the identical program in two different production environments produced normal profiles with only 29% of sequences in common (see section 3.2). This reduces the probability that a mimicry sequence crafted for one installation would work in others. Third, the mimicry attack requires injecting a potentially long sequence of code, which may not be possible in all vulnerabilities. The example given in [74] (shown in Figure 4) requires an additional 128 nullified calls to hide an attack sequence of only 8 system calls. Finally, mimicry attacks may be difficult to implement in practice because of the anomalies generated by the "preamble" of such attacks [35].

However, mimicry attack strategies have become increasingly sophisticated, using automated attack methods including model checking [21], genetic algorithms [34], and symbolic code execution [41]. Although these approaches may not always be reliable in practice, work on persistent interposition attacks shows that the application itself can be used to facilitate mimicry attacks [58]. These results suggest that if an attacker can corrupt program memory, then it is possible to evade virtually any system call-based monitoring system—assuming it is the only defense. It remains unclear how feasible mimicry attacks are on systems with memory corruption defenses such as address-space randomization [60].

Another attack involves crafting sequences that are short enough to avoid producing anomalies, hence exploiting "blind spots" in detection coverage [71, 70]. Any intrusion detection system is a trade-off between false positives and false negatives. In order to reduce potential false positives, the original system used a temporal threshold: Anomalous sequences signaled attacks only if the number of anomalies within a recent time window exceeded a given threshold. This opens the possibility of designing attacks that can stay below the threshold, either by generating very few anomalies or by spreading them out over a long time period. Once again, such attacks require that the attacker inject code containing particular sequences of system calls.

Non-control-flow attacks are yet another way of potentially subverting the system call modeling method. The goal is to manipulate the parameters to system calls without changing the call sequence. Chen et al. demonstrated that there are viable vulnerabilities that can be exploited with this method, for example, by using normal system calls in the sequence to elevate privileges and then overwriting the password file [7]. In a sense, this approach is an extension of the original mimicry attack: instead of nullifying system calls for the purpose of crafting an attack sequence, the manipulated calls become the means of attack directly, without any need to create an attack sequence.

The advent of mimicry and other attacks against the system call modeling approach led to a wealth of research aimed at improving the original method to make it more attack resistant. Many of the extensions discussed below were inspired by a need to address these attacks.

5 Data modeling methods

The methods described in Section 3 and [14, 25] depend only on an enumeration of the empirically observed sequences, or n-grams[2], in traces of normal behavior. Two different methods of enumeration were studied, each of which defines a different model, or generalization, of the data. There was no statistical analysis of these patterns in the original work. As shown in Figure 1, the lookahead pair method constructs a list for each system call of the system calls that follow it at a separation of 0, 1, 2, up to n positions later in the trace. The * character is a wildcard, so the pattern ¡mmap, *, getrlimit¿ specifies that any 3-symbol sequence that begins with mmap and ends with getrlimit will be treated as normal. This method can be implemented efficiently and produced good results on the original data sets. On some data sets, representing the n-grams exactly gave better discrimination than lookahead pairs, although it is more inefficient to implement. Researchers also experimented with variable-length window sizes [49, 78, 11], random schema masks [28]. The correspondence between n-gram representations and finite state automata (FSA) was studied in several papers, including [49, 31]. This work has been extended to more structured representations, but these require additional information such as the execution context, and are discussed in Section 6.

Several methods used statistical machine learning to develop accurate models of normal system call patterns. In an early example, DFA induction was used to learn a FSA that recognized the language of the program traces [40]. In this work, the learning algorithm determined the frequencies with which individual symbols (system calls) occurred, conditioned on some number of previous symbols. Individual states in the automaton represented the recent history of observed symbols, while transitions out of the states specified both which symbols were likely to be produced next and what the resulting state of the automaton would be.

Other machine learning approaches include Hidden Markov Models (HMM) [76, 17], neural networks [18, 10], k-nearest neighbors [47], and Bayes models [42]. Each of these projects was designed to produce a more accurate model, with the goal of reducing false positives. This comes at the cost of more computationally expensive algorithms. In some cases, the algorithms require multiple passes over the entire data set, thus violating the real-time component of the *autonomy* principle. Another limitation of most statistical methods is the assumption of stationarity. This means that the concept of *normal behavior* is assumed not to change while the system is being trained and tested. This assumption violates the *adaptable* principle, a problem addressed in [48] for network traffic, where probabilities were based on the time since the last event rather than on average rate.

Data mining seeks to discover what features are most important out of a large collection of data. This idea was applied to system calls with the goal of discovering a more compact definition of normal than that obtained by simply recording all observed patterns in the training set [45]. Also, by identifying the most important features of such patterns, it was hypothesized that the method would be more likely to generalize effectively.

To summarize, many approaches to modeling system call data have been developed. These range from simple computationally efficient models to more sophisticated approaches that require additional computation. Most innovations have

[2] An n-gram is a sub-sequence of length n taken from a given sequence. Here, the n-gram representation is obtained by sliding a window of length n across the entire sequence.

been aimed either at reducing false-positive rates or at coping with mimicry attacks. It is surprisingly difficult to compare the performance of the different methods in a systematic way. However, one early study [76] concluded that differences among data sets had significantly more impact on results than differences among data modeling methods.

6 Extensions

The idea of using short sequences of system calls for anomaly detection has been extended in several ways. These can be grouped into several broad categories. Here we discuss some of the key advances, without an attempt to exhaustively review all of the research that has been done in these areas.

6.1 Data flow

The original work deliberately discarded any information related to system call parameters. This resulted in a simple, lightweight method. However, a logical extension is to consider the effects of parameters to system calls. Another way of viewing this is that sequences of system calls model *code path flow*, and system call parameters model *data flow*.

Tandon and Chan used a rule-learning system to augment a code-flow anomaly detection system with a data-flow system [72]. They combined rules for sequences of system calls with those for system call arguments. They reported improved attack detection, but at the cost of increased complexity: Their system ran 4-10 times more slowly when arguments were included. Sufatrio and Yap incorporated data flow in the form of a supplied specification for system call arguments [69], and Bhatkar et al. reported that modeling the temporal aspects of data flow in conjunction with control flow further improved detection [6].

Kruegel et al. went one step further, looking only at the arguments and disregarding code flow altogether [43]. They explored several different models for anomaly detection based on system call arguments, including the distribution of string lengths and characters in arguments, Markov models of argument grammar, and explicit enumeration of limited argument options. They demonstrated that their approach is effective against attacks (primarily buffer overflows) and that it has low overhead. Mutz et al. extended this approach by using a Bayesian network to combine the output of the different system call argument models [53].

6.2 Execution context

Apart from system call arguments, there are many additional sources of information associated with system calls that can be used to improve anomaly detection. One of these is the location within the program code from where a system call is issued, which can be determined by the program counter. Sekar et al. first proposed using program counter information to build a FSA of system call sequences [62].

A FSA is a natural model for program code paths; however, inferring a FSA from sequence information alone is difficult. The Hidden Markov Model presented in Warrender et al. is similar to a FSA, and has large learning overheads [76]. Further, in the absence of program counter information, the FSA does not improve detection or reduce false positives dramatically. The key insight in Sekar et al. was that using program counter information in the FSA can overcome these limitations. The states in the FSA are program locations derived from program counters, and the transitions are system calls. Hence the model defines allowable system call transitions from one program location to the next.

A FSA using program counters is a close representation of the true structure of the code, and as such it is able to model loops and both long and short range correlations effectively, unlike the n-gram approach. This results in both increased accuracy of attack detection and reduced false positives [62]. A further benefit is that a FSA model using program counters converges to a stable normal model an order of magnitude faster than an n-gram model.

In addition to the program counter, the call stack is a rich source of information. The VtPath model [12] augments the FSA approach with stack return addresses—each transition of the FSA includes a list of all the return addresses. This is used to generate a virtual path between system calls which can then be additionally checked for anomalies. This additional information improves attack detection and reduces false positives, without incurring additional overhead.

In a similar approach, execution graphs were used to extend simple system call enumeration to a more structured representation [16]. In this approach, the return address pointer is stored with the system call, and this information is used to reconstruct a graph similar to a control flow graph by simply observing the patterns of system calls in a running program. The paper proves that given a set of observed program behavior, the algorithm constructs an execution graph that is consistent with a control flow graph that is obtained through static analysis.

6.3 Static analysis

In the original research, the normal profile was determined by observing running code and recording the sequences of system calls that were executed during normal behavior. However, learning normal at runtime has limitations. First, incompletely learning normal can result in false positives. Second, if learning takes place online, in a vulnerable system, then attacks could potentially be injected into the normal definition. Finally, normal can change, for example, when a system is reconfigured, and hence require relearning.

To address these issues, researchers used static analysis of the program source [73, 22] or binary [20, 5] to develop models of legitimate code paths. This guarantees zero false positives, and no attack injection during learning. The defense mechanism is ready to deploy immediately, without any vulnerable learning period. Such an approach can be effective at defining a normal profile that detects foreign code injection, such as buffer overflow attacks, Trojans, and foreign library calls [74].

However, this approach is limited because attacks often exploit code paths that exist in the program source but are never or rarely used in normal behavior, e.g., a configuration error such as not disabling debugging access. This problem can be mitigated by incorporating more information about the program environment (e.g., configuration, command-line parameters, environment variables) into the static analysis [19]. In general, static and dynamic analysis can complement each other, e.g., by using static analysis to generate a base normal profile, and then incorporating refinements suggested by dynamic analysis [83].

6.4 Other observables

The general idea of profiling program behavior using sequences of operations that indicate code flow is a powerful one that can be applied to many observables other than system calls. For example, Jones and Lin used sequences of library calls, rather than system calls [30]. Similar to the original system call sequence research, Jones and Lin ignored parameters to library calls and only monitored the sequences of calls. They demonstrated that library calls are a feasible observable for anomaly detection, and can be used to detect a variety of attacks, including buffer overflows, denial-of-service attacks and Trojans. In another example, Xu et al. modeled control flow at the level of function calls by inserting *waypoints* into the code at the entry and exit of functions [82], and Gaurev and Keromytis augmented system call monitoring with libc function monitoring [36]. An even more fine-grained approach to control flow is monitor at the level of machine code instructions [1, 63]. This approach can give better attack detection, but at the cost of increased overhead (up to 50% in some cases).

In some domains, other observables may in fact be more suitable. For example, distributed applications may be difficult to monitor comprehensively using sequences of system calls alone. Stillerman et al. showed that in distributed applications a good observable is the messages that are passed across the network [68]. To demonstrate this, they implemented an anomaly detection system for a distributed CORBA application, which consists of many distributed objects that communicate via messages passed over the network. The sequence of messages is a good proxy for object behavior, and can be used to differentiate between normal application behavior and rogue clients.

Further, there may be other sources of information available in other domains that are not available when monitoring running server programs. One example is dynamic execution environments, such as Java, where the interpreter can easily collect a wealth of information about program execution during runtime. Inoue and Forrest showed that Java method invocations are effective observables for anomaly detection [27]. Their approach went beyond attack detection—they used Java's sandboxing security mechanism to implement *dynamic sandboxing*, where minimal security policies are inferred using run-time monitoring.

7 Automated Response

Although there has been extensive research into methods for intrusion detection using system calls, there has been much less work on how to respond to detected anomalies. Most anomaly detection systems produce more alerts than can be handled easily by users or administrators. The problem is compounded when many copies of a system are deployed in a single organization. What is needed, then, are automated responses to detected anomalies. One impediment, however, is the problem of false positives. A binary response, such as shutting down a machine or unauthenticating a user, is unacceptable if there is even a small probability that the response was made in error. If the principle of graduated response is adopted, however, small adjustments can be made continually, and there is less risk of damaging the system needlessly or enabling a denial of service.

The first effort to couple an automated response to system call anomalies was a Linux kernel extension called pH [64, 65]. pH detects anomalies using lookahead pairs. Instead of killing or outright blocking the behavior of anomalously behaving processes, it delays anomalous system calls: Isolated anomalies are delayed imperceptibly, and clustered anomalies are delayed exponentially longer. Real attacks tend to generate large delays (on the order of hours or days)]. Because most network connections have built-in time outs, this response automatically blocks many attacks; further, it gives administrators time to intervene manually. Many false alarms produce a small number of isolated anomalies, and pH responds with a proportionally small delay that is usually imperceptible to the user. In the rare case of a false positive causing a long delay, a simple override was provided for the user or administrator.

Although pH was the first system to use delays as a response to detected anomalies, delay-based strategies have been used in other defenses, especially in networking and for remote login interfaces (e.g., to prevent online dictionary attacks). For example, Williamson observed that unusually frequent outgoing network requests could signal an anomaly, and that the damage caused by such behavior could be mitigated simply by reducing the rate at which new network connections could be initiated [80]. This technology became part of HP's ProCurve Network Immunity Manager [26], and it was extended to include incoming connections and more types of network connections in [4]. This idea of slowing down a computation or communication is often referred to as *throttling* or *rate limiting*. It has been studied extensively in the networking community, for example in active networks [23], Domain Name Service [81], Border Gateway Protocol [33, 32], and peer-to-peer networks [2].

A commercial implementation of system-call anomaly detection, however, implemented another response strategy. Sana Security's Primary Response used a layered approach. The first layer was a mechanism to explicitly prevent code-injection, in all forms, covering a large class of common attacks and preventing subversion through mimicry attacks. The second layer blocked anomalous system calls that manipulated the file system. This can prevent many non-code-injection attacks and most applications are more robust to failures of file system calls than other system calls. Further, Primary Response also profiled parameters to file-related system calls, and hence could use that additional information to further reduce false positives and prevent non-control-flow attacks.

8 Summary and Conclusions

Over the past decade, we have witnessed continual evolution of new platforms and new forms of attack, including the advent of email viruses, spyware, botnets, and mutating malware, just to name a few. Research on system-call monitoring matured over this time period as well, and many variations of the original method have been explored. In the previous sections we highlighted representative examples of this work, examples that we feel illustrate the breadth and depth of the method. Despite dramatic changes in today's computing environments and applications, system-call monitoring remains a fundamental technique underlying many current projects, e.g., [77, 52]. This is remarkable, although it is optimistic to expect system-call monitoring per se to remain an active and exciting research frontier indefinitely. Both the threats and the defenses against them will continue to evolve, likely migrating to higher application layers and to lower levels, such as on-chip attacks in multi-core architectures.

The design principles articulated in Section 2, those principles inspired by living systems, are potentially of more lasting significance. These include: generic mechanisms, adaptability, autonomy, graduated response, and diversity. Some of these principles have been adopted widely (diversity), some remain controversial (graduated response and adaptability), and some have been largely ignored (generic mechanisms and autonomy). Taken together, these principles constitute a hypothesis about what properties are required to protect computers and their users.

The design principles guided nearly all of our implementation decisions. This is one example of how the study of computer security can be more scientific than an ad hoc collection of specific instances. By articulating a hypothesis, and then designing the simplest possible experiment to test that hypothesis, i.e. that anomaly detection could protect privileged processes against important classes of attack, we were able to demonstrate that short sequences of system calls are a good discriminator between normal and malicious behavior. Rather than focusing only on producing an artifact that worked, we set out to understand what approaches could be used for effective defenses. Indeed, in the beginning we tested system call sequences only as a base case, expecting that enhancements would be required for the method to work. Instead of studying the enhancements, however, we found ourselves analyzing why the experiment succeeded.

A key component of our approach was designing repeatable experiments. This allowed others to confirm our results and test variations against the original system. Although this point seems obvious in retrospect, it was unique at the time. Our experiments were repeatable because our system design was comprehensible, and we published both our data sets and the software prototypes in the public domain. This enabled other groups to replicate our results, use the data sets for their own experiments, use our code to devise attacks against the method, and so forth.

Repeatable experiments are crucial to putting computer security on sounder footing. However, they are not sufficient. Careful comparisons between competing methods are also important, and this has been much more difficult for the field to achieve. Although public data sets and prototypes help this effort, it is still extraordinarily difficult to conduct comparisons carefully. There are several reasons for this: (1) Environments are complex; (2) Results depend heavily on data inputs; and (3) Metrics emphasize breadth of coverage. The complexity of modern computing environments poses a serious challenge to replicating results and comparing results from multiple experiments. Is is surprisingly difficult to document precisely the conditions under which an experiment is run, and seemingly trivial differences in system configuration can affect the outcome of an experiment dramatically. This problem is even more challenging in networking experiments than it is for single hosts. Further, we discovered that for system calls the outcome of our comparisons depended heavily on which program traces we selected for the comparison [76]. It was easy to skew the results of our comparative studies by appropriate (or inappropriate) choice of data sets.

Most systems are judged by their ability to defend against as many attacks as possible. This leads to system designs that are optimized for broad coverage and corner cases, which typically lead to complex algorithms and implementations. As Sections 5 and 6 show, there is a nearly limitless variety of ways that the original idea of short sequences of system calls can be made more complex. To our knowledge, not a single paper has been published that proposes a simpler approach to monitoring system calls. Tuning up other people's methods so they behave optimally requires considerable skill, effort, and discipline; in contrast, a system that highlights one key idea can easily be evaluated because there are fewer "knobs" to adjust—evaluations in different contexts are thus inherently easier to compare.

What began as a simple insight inspired by biology has grown into a robust and diverse field of research. We are excited at the progress and directions that this research has taken. Attacks on the ideas have led to creative new methods that make the original approach much more robust to subversion, and various other improvements and developments have resulted in a far better protection system than we could have hoped to see over a decade ago. This has validated some of the principles elucidated in section 2, but we see much scope for extending the research to investigate those principles in greater depth. We hope that this paper illustrates how inspirational the biological analogy can be, and encourages others to explore those principles so that this continues to be a vibrant ever-growing area of research.

9 Acknowledgments

The authors gratefully acknowledge the many people who encouraged and assisted us during the development of the original system call project. In particular, we thank Dave Ackley, Tom Longstaff, and Eugene Spafford. Jed Crandall, Dave Evans, ThanhVu Nguyen, and Eugene Spafford made many helpful suggestions on this manuscript.

The original project was partially funded by the National Science Foundation (NSF) IRI-9157644, Office of Naval Research N00014-95-1-0364, and the Defense Advanced Research Projects Agency N00014-96-1-0680. SF acknowledges NSF (CCF 0621900, CCR-0331580), Air Force Office of Scientific Research MURI grant FA9550-07-1-0532, and the Santa Fe Institute. AS acknowledges NSERC's Discovery program and MITACS.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity: Principles, implementations and applications. In *Proceedings of ACM Computer and Communications Security*, November 2005.

[2] M. K. Aguilera, M. Lillibridge, and X. Li. Transaction rate limiters for peer-to-peer systems. *IEEE International Conference on Peer-to-Peer Computing*, 0:3–11, 2008.

[3] D. Anderson, T. Frivold, and A. Valdes. Next-generation intrusion detection expert system (nides): A summary. Technical Report SRI-CSL-95-07, Computer Science Laboratory, SRI International, May 1995.

[4] J. Balthrop. Riot: A responsive system for mitigating computer network epidemics and attacks. Master's thesis, The University of New Mexico, Albuquerque, NM, 2005.

[5] S. Basu and P. Uppuluri. *Proxi-Annotated Control Flow Graphs: Deterministic Context-Sensitive Monitoring for Intrusion Detection*, pages 353–362. Springer, 2004.

[6] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *In Proc. IEEE Symposium on Security and Privacy*, pages 48–62, 2006.

[7] S. Chen, J. Xu, and E. C. Sezer. Non-control-data attacks are realistic threats. In *14th Annual Usenix Security Symposium*, Aug 2005.

[8] R. Danyliw and A. Householder. Cert advisory ca-2001-19: Code red worm exploiting buffer overflow in iis indexing service dll. Website, 2001. http://www.cert.org/advisories/CA-2001-19.html.

[9] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13:222–232, 1987.

[10] D. Endler. Intrusion detection: applying machine learning to solaris audit data. In *In Proc. of the IEEE Annual Computer Security Applications Conference*, pages 268–279. Society Press, 1998.

[11] E. Eskin, W. Lee, and S. J. Stolfo. Modeling system calls for intrusion detection with dynamic window sizes,. In *Proceedings of DARPA Information Survivability Conference and Exposition II (DISCEX II)*, Anaheim, CA, 2001.

[12] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.

[13] G. Fink and K. Levitt. Property-based testing of privileged programs. In *Proceedings of the 10th Annual Computer Security Applications Conference*, page 154163, Dec. 1994.

[14] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 120, Washington, DC, USA, 1996. IEEE Computer Society.

[15] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri. Self-nonself discrimination in a computer. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, Los Alamitos, CA, 1994. IEEE Computer Society Press.

[16] D. Gao, M. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pages 318–329, October 2004.

[17] D. Gao, M. K. Reiter, and D. Song. Behavioral distance measurement using hidden markov models. In D. Zamboni and C. Kruegel, editors, *Research Advances in Intrusion Detection*, LNCS 4219, pages 19–40, Berlin Heidelberg, 2006. Springer-Verlag.

[18] A. Ghosh and A. Schwartzbard. A study in using neural networks for anomaly and misuse detection. In *Proceedings of the 8th USENIX Security Symposium*, 1999.

[19] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller. Environment-sensitive intrusion detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID '05)*. Springer, Sep 2005.

[20] J. T. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *11th Usenix Secutirty Symposium*, August 2002.

[21] J. T. Giffin, S. Jha, and B. Miller. Automated discovery of mimicry attacks. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID '06)*, Sep 2006.

[22] R. Gopalakkrishna, E. H. Spafford, and J. Vitek. Efficient intrusion detection using automaton inlining. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 18–31, 2005.

[23] A. Hess, M. Jung, and G. Schfer. Fidran: A flexible intrusion detection and response framework for active networks. In *In Symposium on Computers and Communications (ISCC2003*, 2003.

[24] S. A. Hofmeyr and S. Forrest. Architecture for an artificial immune system. *Evolutionary Computation Journal*, 8(4):443–473, 2000.

[25] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3), 1998.

[26] HP. Immunity manager. Website. http://www.hp.com/rnd/pdfs/ProCurve_Network_Immunity_Manager1_0.pdf.

[27] H. Inoue and S. Forrest. Inferring java security policies through dynamic sandboxing. In *Proceedings of the 2005 International Conference on Programming Langauges and Compilers (PLC '05)*, June 2005.

[28] H. Inoue and A. Somayaji. Lookahead pairs and full sequences: a tale of two anomaly detection methods. In *Proceedings of the 2nd Annual Symposium on Information Assurance*, June 2007.

[29] A. Jones and S. Li. Temporal signatures for intrusion detection. In *Seventeenth Annual Computer Security Applications Conference, 10-14 Dec. 2001, New Orleans, LA, USA*, pages 252–61. Los Alamitos, CA, USA : IEEE Computer Society, 2001, 2001.

[30] A. Jones and Y. Lin. Application intrusion detection using language library calls. In *Proceedings of the 17th Annual Computer Security Applications Conference*, New Orleans, Louisiana, December 10–14, 2001.

[31] B. M. K Wee. Automatic generation of finite state automata for detecting intrusions using system call sequences. In V. G. et al., editor, *Mathematical Methods, Models, and Architectures for Network Security Systems (MMM-ACNS)*, pages 206–216. Springer-Verlag Berlin Heidelberg, 2003.

[32] J. Karlin, J. Rexford, and S. Forrest. Pretty good bgp: Improving bgp by cautiously adopting routes. In *Proc. of the 2006 International Conference on Netowrk Protocols (CNP)*, 2006.

[33] J. Karlin, J. Rexford, and S. Forrest. Autonomous security for autonomous systems. *Computer Networks*, 52:29082923, 2008.

[34] H. G. Kayacik, M. Heywood, and N. Zincir-Heywood. On evolving buffer overflow attacks using genetic programming. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1667–1674, New York, NY, USA, 2006. ACM.

[35] H. G. Kayacik and A. N. Zincer-Heywood. On the contribution of preamble to information hiding in mimicry attacks. In *Proceedings of the IEEE Symposium on Security in Networks and Distributed Systems - SSNDS'2007*, 2007.

[36] G. S. Kc and A. D. Keromytis. e-nexsh: Achieving an effectively non-executable stack and heap via system-call policing. In *In Proceedings of the Annual Computer Security Applications Conference(ACSAC*, 2005.

[37] J. O. Kephart, G. B. Sorkin, W. C. Arnold, D. M. Chess, G. J. Tesauro, and S. R. White. Biologically inspired defenses against computer viruses. In *IJCAI '95*. International Joint Conference on Artificial Intelligence, 1995.

[38] J. O. Kephart, S. R. White, and D. M. Chess. Computers and epidemiology. *IEEE Spectrum*, 30(5):20–26, 1993.

[39] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in priviledged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, page 134144, Dec. 1994.

[40] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Software*, 14(5):35–42, September/October 1997.

[41] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *14th Annual Usenix Security Symposium*, Aug 2006.

[42] C. Kruegel, D. Mutz, W. Robertson, and F. Valeur. Bayesian event classification for intrusion detection. In *In 19th Annual Computer Security Applications Conference, LasVegas*, page 14. IEEE Computer Society, 2003.

[43] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceeding of ESORICS 2003*, pages 326–343. Springer-Verlag Berlin Heidelberg, oct 2003.

[44] T. Lane and C. E. Brodley. Temporal sequence learning and data reduction for anomaly detection. *ACM Transactions on Information and System Security*, 2(3):295–331, 1999.

[45] W. Lee and S. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[46] T. Li, J. Ding, X. Liu, and P. Yang. A new model of immune-based network surveillance and dynamic computer forensics. In K. C. L. Wang and Y. Ong, editors, *Proceedings of the The First International Conference on Natural Computation*, page 804813. IEEE, Springer-Verlag Berlin Heidelberg, 2005.

[47] Y. Liao and V. R. Vemuri. Use of k-nearest neighbor classifier for intrusion detection. *Computers & Security*, 21(5):439–448, 2002.

[48] M. Mahoney and P. Chan. Learning nonstationary models of normal network traffic for detecting novel attacks. In *Proceeding of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2000.

[49] C. Marceau. Characterizing the behavior of a program using multiple-length n-grams. In *Proceedings of the New Security Paradigms Workshop 2000*, Cork, Ireland, Sept. 19–21, 2000. Association for Computing Machinery.

[50] C. Michael and A. Ghosh. Two state-based approaches to program-based anomaly detection. In *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC'00)*, New Orleans, LA, December 11–15 2000.

[51] W. H. Murray. The application of epidemiology to computer viruses. *Computers & Security*, 7:139–150, 1988.

[52] D. Mutz, W. Robertson, G. Vigna, and R. Kemmerer. Exploiting Execution Context for the Detection of Anomalous System Calls. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 1–20, Gold Coast, Australia, September 2007.

[53] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, 2006.

[54] D. Q. Naiman. Statistical anomaly detection via httpd data analysis. *Computational Statistics and Data Analysis*, 45:51–67, 2004.

[55] M. Newman, S. Forrest, and J. Balthrop. Email networks and the spread of computer viruses. *Physical Review E*, 66(035101), 2002.

[56] NIST. National vulnerability database. Website, 2008.

[57] T. Oda and T. White. Immunity from spam: An analysis of an artificial immune system for junk email detection. In C. Jacob, M. Pilat, P. Bentley, and J. Timmis, editors, *Artificial Immune Systems*, Lecture Notes in Computer Science, pages 276–289, Germany, 2005. Springer-Verlag.

[58] C. Parampalli, R. Sekar, and R. Johnson. A practical mimicry attack against powerful system-call monitors. In *ASIACCS '08: Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 156–167, New York, NY, USA, 2008. ACM.

[59] R. Pastor-Satorras and A. Vespignani. Epidemic spreading in scale-free networks. *Physical Review Letters*, 86(14):3200–3203, Apr. 2001.

[60] PaX Team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt.

[61] T. H. Ptacek and T. N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Jan 1998.

[62] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[63] M. S. Sharif, K. Singh, J. Giffin, and W. Lee. Understanding precision in host based intrusion detection. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 21–41, 2007.

[64] A. Somayaji. *Operating System Stability and Security through Process Homeostasis*. PhD thesis, University of New Mexico, 2002.

[65] A. Somayaji and S. Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, August 14–17, 2000.

[66] E. H. Spafford. Computer viruses—a form of artificial life? In C. G. Langton, C. Taylor, J. D. Farmer, and S. Rasmussen, editors, *Artificial Life II*, pages 727–745. Addison-Wesley, Redwood City, CA, 1992.

[67] E. H. Spafford. Virus. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

[68] M. Stillerman, C. Marceau, and M. Stillman. Intrusion detection for distributed applications. *Communications of the ACM*, 42(7):62–69, July 1999.

[69] Sufatrio and R. H. C. Yap. Improving host-based ids with argument abstration to prevent mimicry attacks. In *Proceedings of the International Sypmosium on Recent Advances in Intrusion Detectino (RAID)*, pages 146–164, 2006.

[70] K. M. C. Tan, K. S. Killourhy, and R. A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. In *Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID '02)*, 2002.

[71] K. M. C. Tan and R. A. Maxion. "Why 6?" defining the operational limits of stide, an anomaly-based intrusion detector. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 188, Washington, DC, USA, 2002. IEEE Computer Society.

[72] G. Tandon and P. Chan. On the learning of system call attributes for host-based anomaly detection. *International Journal on Artificial Intelligence Tools*, 15(6):875–892, 2006.

[73] D. Wagner. *Static Analysis and Computer Security: New Techniques for Software Assurance*. PhD thesis, University of California at Berkeley, 2000.

[74] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–169, 2001.

[75] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 255–264, New York, NY, USA, 2002. ACM Press.

[76] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145, Los Alamitos, CA, 1999. IEEE Computer Society.

[77] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 461–476, 2005.

[78] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Proceedings of the 2000 Recent Advances in Intrusion Detection*, page 110129, October 2000.

[79] P. D. Williams, K. P. Anchor, J. L. Bebo, G. H. Gunsch, and G. D. Lamont. Cdis: Towards a computer immune system for detecting network intrusions. In W. Lee, L. Me, and A. Wespi, editors, *Fourth International Symposium, Recent Advances in Intrusion Detection*, pages 117–133, Berlin, 2001. Springer.

[80] M. M. Williamson. Throttling viruses: Restricting propagation to defeat malicous mobile code. In *Proceedings of ACSAC Security Conference*, Las Vegas, Nevada, Dec. 2002.

[81] C. Wong, S. Bielski, A. Studer, and C. Wang. On the effectiveness of rate limiting mechanisms. In *Proc. 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, 2005.

[82] H. Xu, W. Du, and S. J. Chapin. Context sensitive anomaly monitoring of process control flow to detect mimicry attacks and impossible paths. In *In Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID*, pages 21–38. Springer, 2004.

[83] L. Zhen, S. M. Bridges, and R. B. Vaughn. Combining static analysis and dynamic learning to build accurate intrusion detection models. In *Proceedings of the 3rd IEEE International Workshop on Information Assurance*, March 2005.