Analysis of Computer Intrusions Using Sequences of Function Calls

Sean Peisert, *Student Member*, *IEEE*, Matt Bishop, *Member*, *IEEE*, Sidney Karin, *Member*, *IEEE*, and Keith Marzullo

Abstract—This paper demonstrates the value of analyzing sequences of function calls for forensic analysis. Although this approach has been used for intrusion detection (that is, determining that a system has been attacked), its value in isolating the cause and effects of the attack has not previously been shown. We also look for not only the presence of unexpected events but also the absence of expected events. We tested these techniques using reconstructed exploits in `su`, `ssh`, and `lpr`, as well as proof-of-concept code, and, in all cases, were able to detect the anomaly and the nature of the vulnerability.

Index Terms—Security, forensic analysis, logging, auditing, intrusion detection, anomaly detection, management, design, unauthorized access (for example, hacking).

1 Introduction and Motivation

"It is, of course, a trifle, but there is nothing so important as trifles."

—Sir Arthur Conan Doyle, "The Man with the Twisted Lip"

The Strand Magazine (1891)

Forensic analysis is the process of understanding, recreating, and analyzing events that have previously occurred [1]. The problem of computer forensics is not simply finding a needle in a haystack: It is finding a needle in a stack of needles. Given a suspicion that a break-in or some other "bad" thing has occurred, a forensic analyst needs to localize the damage and determine how the system was compromised. With a needle in a haystack, the needle is a distinct object. In forensics, the point at which the attacker entered the system can be very hard to ascertain, because in audit logs, "bad" events rarely stand out from "good" ones.

In this paper, we demonstrate the value of recording function calls to forensic analysis. In particular, we show that function calls are a level of abstraction that can often make sense to an analyst. Through experiments, we show that the technique of analyzing sequences of function calls that deviate from previous behaviors gives valuable clues about what went wrong.

Forensic data logged during an intrusion should be detailed enough for an automated system to flag a potentially anomalous behavior and descriptive enough for a forensic analyst to understand. Although collecting as much data as possible is an important goal [2], a trace of machine-level instructions, for example, may be detailed enough for automated computer analysis, but is not descriptive enough for a human analyst to interpret easily.

There has been considerable success in capturing the system behavior at the system call (sometimes called *kernel call*) level of abstraction. This has been applied to several computer security problems, including intrusion detection [3], [4], [5], [6], [7], forensics [8], [9], confinement [10], [11], and illicit information flow [12]. Even the popular Basic Security Module (BSM) [13] and its cross-platform derivatives are largely based on a subset of a given system's kernel calls. All users, whether authorized or not, must interact with the kernel and, therefore, use system calls to perform privileged tasks on the system. In addition, kernel calls are trivial to capture and are low-cost, high-value events to log, as opposed to the extremes of logging everything (such as all machine instructions) or logging too little detail for effective forensic analysis (such as syslog). Capturing behaviors represented at the system call abstraction makes intuitive sense: Most malicious things an intruder will do use system calls. Nonetheless, extending the analysis of behaviors to include more data than system calls by collecting function calls can produce information useful to a human [14] without generating impractical volumes of data. Though function call tracing is not new,[1] we analyze sequences of function calls in a way that results in an improved forensic analysis, which we believe *is* new.

Logging all function calls can generate a huge amount of data. Function calls capture forensically significant events that occur in both user space and kernel space (system calls are essentially protected function calls). In our experiments, between 0.5 percent and 5 percent of function calls recorded in behaviors are system calls. Thus, the amount of audit data increases from 20 to 200 times as compared to recording only system calls. This additional data makes it much easier to determine when something wrong took place, what it was exactly, and

- *S. Peisert, S. Karin, and K. Marzullo are with the Department of Computer Science and Engineering, University of California, San Diego, 9500 Gilman Drive #0404, La Jolla, CA 92093-0404. E-mail: {peisert, karin, marzullo}@cs.ucsd.edu.*
- *M. Bishop is with the Department of Computer Science, University of California, Davis, One Shields Ave., Davis, CA 95616-8592. E-mail: bishop@cs.ucdavis.edu*

Manuscript received 5 Sept. 2006; revised 17 Jan. 2007; accepted 13 Mar. 2007; published online 28 Mar. 2007.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0126-0906.
Digital Object Identifier no. 10.1109/TDSC.2007.1003.

1. For example, it was used for profiling in 1987 [15] and, more recently, in intrusion detection [16], [17], [18].

how it happened. Additionally, as we will describe, the increase in the amount of data *recorded* does not necessarily translate into a proportional increase in the amount of data necessary for a human to *audit*.

Our approach comes partially from intrusion detection. The techniques need to be modified to be useful for forensic analysis, but as we will show here, they have good utility. In Section 7, we discuss some of the specific similarities and differences between the goals, needs, and techniques used in forensic analysis and intrusion detection.

We demonstrate the utility of our approach by giving a methodology for examining sequences of function calls and showing experiments that result in manageable amounts of understandable data about program executions. In many instances, our techniques offer an improvement over existing techniques. We also make recommendations about practical implementations and use of this process at the end of the paper.

Our methods follow four principles [2]: They collect function calls within the user space, they collect data at runtime, they look at the impact of the user's actions on the system, and they process and present the data in a meaningful way.

2 Background

2.1 Anomaly Detection

There are two common techniques used for intrusion detection [19]. *Anomaly detection* involves looking for a statistical deviation from a safe corpus of data. *Misuse detection* involves analyzing whether events or scenarios fit a predefined model. Similarly, the analysis phase can be performed either in real time or after the fact.

Previous work [3], [20], [21] used sequences of system calls to understand anomalous system activity. This approach allowed the experimenters to delay calls in response to suspected intrusions to perform analyses in real time [21], [22]. To simplify the analysis and reduce the amount of data collected, the early implementations logged only those system calls made by privileged programs (setuid to root or setgid to wheel). However, all of the implementations analyzed only the patterns of system calls and not the parameters or return values.

The original analysis techniques were examples of *instance-based machine learning*, which generally involves comparing new instances of data whose class is unknown with existing instances whose class is known [23]. In this case, the experimenters compared windows of system calls of a specific size between the test data and data known to be nonanomalous using *Hamming distances*. The original research was done over a number of years, and the definition of *anomaly* changed over time. At some points, an anomaly was flagged when a Hamming distance greater than zero appeared. At other times, an anomaly was flagged when Hamming distances were large or when many mismatches occurred. In 1999, the experimenters revisited their original data with new analysis methods [24]. Some of the new methods included looking at rare sequences in addition to unknown sequences using data mining techniques and using Hidden Markov Models (HMMs).

The new analyses provided few new results and, in fact, none stood out as superior. This suggests that those methods requiring large amounts of computational power (such as HMMs) may not be worthwhile. Further, an optimal window size was not determined. The window size of six used in the experiments was shown to be an artifact of the data used and not a generally recommended one [25]. The experimenters later used Java methods rather than system calls, but the work used only individual methods, not sequences of methods, and was applied to dynamic sandboxing, not forensics [26].

Data mining has been used for anomaly detection [27], [28], but has used only coarsely grained data. Approaches using system calls and expanded techniques have subsequently been explored, with good success [5], [29].

2.2 Forensic Analysis

Logging, one of two primary components of forensic analysis, is the recording of data that can be useful in the future for understanding past events. *Auditing*, the other primary component, involves gathering, examining, and analyzing the logged data to understand the events that occurred during the incident in question [1]. In practice, forensic analysis generally involves locating suspicious objects or events and then examining them in enough detail to form a hypothesis as to their cause and effect. Data for forensic analysis can be collected from a virtual machine during a deterministic replay [30], as long as the overhead for the nondeterministic event logging is acceptable. A highly specialized hardware [31], [32] might make nondeterministic event logging practical.

A practical issue in forensics is the trade-off between accuracy and the amount of data recorded. A forensic solution at one extreme [2] is to record everything reflecting an intruder's intentions. This would include all memory accesses explicitly made in an intruder's program, rather than those added as intermediate storage by the compiler. The other end of the spectrum is to record very high-level (and unstructured [33]) data, such as syslog messages or data that is focused in one particular area. Examples of this include file system data from *Tripwire* and *The Coroner's Toolkit* or connection data from *TCP Wrappers*.

Gross [34] exhaustively researched the usable data and analysis techniques from unaugmented systems. It is likely that to do better, systems in the future will need to be enhanced to capture additional forensic data.

System call traces can also be used for forensic analysis. BackTracker [9] is a forensic tool that captures and uses system call data for analyzing problems on the process and file system level. An analyst provides a filename, inode, or process ID as input to BackTracker. Then, based on previously recorded system calls and some assumptions about system call dependencies, BackTracker generates graphical traces of system events that have affected or have been affected by the file or process given as input. However, an analyst using Backtracker may not know what input to provide, since suspicious files and process IDs are not easy to discover when the analysis takes place long after the intrusion. Unfortunately, Backtracker does not help identify the starting point; it was not a stated goal of Backtracker. There have been useful improvements to the "pruning" function of the BackTracker's event-dependency graph [35], which makes the event graphs easier to analyze, but since the process of invoking BackTracker is unchanged, the same limitations on its usefulness still apply.

BackTracker has also been extended [36] to use alerts from network intrusion detection systems to provide additional clues to investigate "multihop attacks" and to enable the creation of event graphs that include events across multiple hosts. This added information is useful, but since the alerts are from network intrusion detection systems instead of host-based intrusion detection systems, BackTracker is still limited in its ability to provide clues to an analyst about what to give as input to generate the event graphs.

Forensix [8] is a tool that collects system calls similar to the way BackTracker does, but rather than generating an event graph, it uses a database query system to answer specific requets that an analyst might have, such as, "Show me all processes that have written to this file." It also contained a feature that streamed audit data off of the host machine in real time to append-only storage on a separate machine in a different security domain. Forensix had similar constraints as BackTracker, however. A forensic analyst, for example, has to independently determine which files might have been written to by an intruder's code.

Data from intrusion detection systems has been proposed for use as forensic evidence, but the papers containing those proposals focus on legal admissibility [37] and on using intrusion detection systems simply because that the data is already collected in real time [38] and not on the utility of the data collected.

Previous research has also been performed to understand the limits of auditing in general [39] and auditing for policy enforcement [40], [41]. However, neither of these previous research efforts were aimed at presenting *useful* information to a human analyst. They were not specifically aimed at forensic analysis but had different goals such as process automation. Other work [42] evaluated the effect of using different audit methods for different areas of focus (attacks, intrusions, misuse, and forensics) with different temporal divisions (real time, near real time, periodic, or archival), but again, the results focused primarily on performance rather than the forensic value to a human.

3 Methods

3.1 Anomaly Detection Using Sequences of Function Calls

With postmortem analysis, a system can record more data, and analysts can examine the data more thoroughly than in real-time intrusion detection. Ideally, the analysts have available a complete record of execution, which enables them to classify sequences as "rare" or "absent." A real-time intrusion detection system, on the other hand, must classify sequences without a complete record because not all executions have terminated. Hence, if a sequence generally occurs near the end of an execution, classifying them as "absent" in the beginning would produce a false positive.

Were we to capture events for forensic analysis on replay, the state of the replayer would need to match that of the system when the events occurred in order for the capture to provide the same information—and this is very difficult to guarantee. We are capturing events at runtime in accordance with the principle stating that we are more likely to obtain genuine data not distorted by an intruder or a change of state in the hardware or software when we capture the data as the event is occurring [2].

Our anomaly detection techniques use the instance-based learning method mentioned previously [3], [20], because it is simple to implement and comparable in effectiveness to the other methods. However, rather than system calls, as were previously used, we do so using function calls and sometimes also indications of the points at which functions return (hereafter, "returns"). The reason for also including function returns is that we determined that they are also useful and sometimes necessary to finding anomalies in the sequences. Currently, our instance-based learning uses a script to separate the calls into sequences of length k, with k being from the set $1, \ldots, 20$. We calculate the Hamming distance between all "safe" sequences and the new sequences for several different values of k.

The sequence length is important for anomaly detection [6], [7], [25], [43]. However, the length to consider depends on a number of factors. If k is small, the sequences may not be long enough for an analyst to separate normal and anomalous sequences. Also, short sequences can be so common that they may be in the "safe" corpus even if they are part of an anomalous sequence at another time. On the other hand, with instance-based learning techniques, the number of distinct sequences of length k increases exponentially as k increases linearly. Also, the number of anomalous sequences that a human analyst has to look at grows as well, though not exponentially. Through experimentation, we discovered that values of k larger than 10 generally should be avoided.

Generally, since our analysis is postmortem, we are more concerned about the human analyst's efficiency and effectiveness than with computing efficiency. By using automated parallel preprocessing that presents options for several values of k, a forensic analyst can decide which sequence lengths to use. We show the effects of choosing different values of k in this paper, but do not claim that a particular value of k is ideal. Ultimately, given that forensic analysis will remain a lengthy iterative process, the sequence length parameter is one that a human analyst will choose and vary according to the situation being analyzed.

3.2 Forensic Methods

Anomaly detection is the foundation for our forensic analysis. The anomaly detection process flags anomalous executions and presents the relevant data for further study. Whereas the anomaly detection process is automated, the forensic process involves a human. Though forensic analysis will undoubtedly be more easily automated in the future, automation is currently hard.

To begin the forensic analysis, a human analyst first decides which sequence length to choose to investigate further. Given that we calculate the number of differing sequences for several values of k, the analyst should choose one that is manageable to look at but in which anomalies are present.

4 Experiments and Results

We compared sequences of function calls from an original (nonanomalous) program with several versions of that same program modified to violate some security policy. Our goal was to determine how readily the differences could be detected and what we could learn about them. We chose the

experiments as examples of important and common classes of exploits. They recreated common critical hacks and exploits that have occurred on computer systems, including spyware,[2] buffer overflows [44], race conditions [45], and Trojan horses [46]. Indeed, Somayaji [21] considered some of these in his dissertation. They cover all classes of flaw domains as enumerated in the Research In Secured Operating Systems (RISOS) [47] and Protection Analysis (PA) [48] reports, except for the "inconsistent parameter validation" class from RISOS. The latter requires looking exclusively at call parameters and not at sequences of calls themselves, and this requires analysis techniques outside of the scope of our work. It is possible that a large enough collection of examples with enough overlapping coverage of the flow domains might be sufficient to analyze not just the attacks specified but any other attack in the same flaw domain(s) as the specified attacks. At this time, we do not assert this, but instead use the fact that these examples cover all flaw domains in the two reports as a means of demonstrating the effectiveness of our approach.

The following is a list of each experiment and the flaw domain[3] that it covers:

1. Omitting or ignoring authentication (Section 4.1):
   - RISOS: inadequate identification/authorization/authentication
   - PA #2: improper validation (of operands, queue mgmt. depend)
   - PA #4: improper choice of operand or operation
2. Spyware (Section 4.2):
   - RISOS: implicit sharing of privileged/confidential data
   - PA #1a: improper choice of initial protection domain
3. Ignoring permissions (Section 4.3):
   - RISOS: inadequate identification/authorization/authentication
   - RISOS: asynchronous-validation/inadequate-serialization
   - PA #1e: improper de-allocation or deletion
   - PA #3a/b: improper synchronization (indivisibility + sequencing)
4. Buffer overflow (Section 4.4):
   - RISOS: violable prohibition/limit
   - RISOS: incomplete parameter validation
   - PA #1b: improper isolation of implementation detail
   - PA #1c: improper change
   - PA #2: improper validation (of operands, queue mgmt. depend.)
5. Trojan horse (Section 4.5):
   - RISOS: Exploitable logic error
   - PA #1d: Improper naming.

2. We define spyware to be a program or a modification to an existing program that quietly gathers information and covertly shares it with an attacker either locally or via the network.

3. The organization that we use for the PA flaws corresponds to the revised hierarchy outlined by Neumann [49], not the organization in the original PA paper.

We ran the first four experiments on an Intel-based uniprocessor machine running FreeBSD 5.4. In those experiments, we began by using Intel's dynamic instrumentation tool *Pin* [50] to instrument the original and modified versions of the programs to record all function calls made. In the last experiment, we used the `ltrace` tool on an Intel-based uniprocessor machine running Fedora Core 4. The `ltrace` tool captures only dynamic library calls, rather than user function calls, but unlike the *Pin* tool, is type aware and therefore enables analysis of parameters and return values. System calls are captured by both instrumentation methods.[4]

To create a database of calls to test against, we ran unmodified versions of the binaries one or more times. For example, for our experiments with `su` below, one of the variations that we tested included successful and unsuccessful login attempts.

In the experiments, some sequences appeared multiple times in a program's execution. We refer to the number of *distinct sequences* in an execution, counting multiple occurrences only once. (The *total number* of sequences is simply the total number of calls minus the length of the sequence k plus 1.) When we compare the safe program's execution to the modified program's execution, we refer to the sequences appearing only in one execution and not the other as the *different sequences*. The relevant numbers are the number of *total different sequences* in each version and the number of *distinct different sequences* in each version, where, again, multiple occurrences are counted only once.

4.1 Omitting and Ignoring Authentication

4.1.1 su Experiment 1

Our first experiment illustrates a simple manually constructed anomaly. We compared the execution of a normal unaltered version of the UNIX `su` utility with one in which the call to `pam_authenticate` was removed, thus removing the need for a user to authenticate when using su.

Fig. 1a shows the number of distinct sequences of function calls for the executions of the two `su` programs. Fig. 1b shows the number of sequences of function calls appearing only in one version of the execution but not the other. The `su`-mod curve in Fig. 1b quickly jumps from 7 when $k = 2$ to 46 when $k = 4$ and 93 when $k = 6$. This pattern is typical and emphasizes why larger values of k can be a hindrance to understanding the data. Ninety-three sequences of length 6 is a lot of data to analyze visually. Although shorter sequences may fail to highlight intrusions, longer sequences can present overwhelming amounts of data to a forensic analyst.

Choosing $k = 4$ somewhat arbitrarily, here is the sequence information for the original and modified versions of `su`:

$k = 4$	number of sequences in each execution
`su`-original	37142 (2136 distinct)
`su`-modified	8630 (1812 distinct)

The discrepancy between the numbers of sequences in each version is sufficient to determine that *something* is

4. We capture both the system call and `libc` interface to the system call, so we can determine when a function call to `libc` just calls a system call and when it does not.

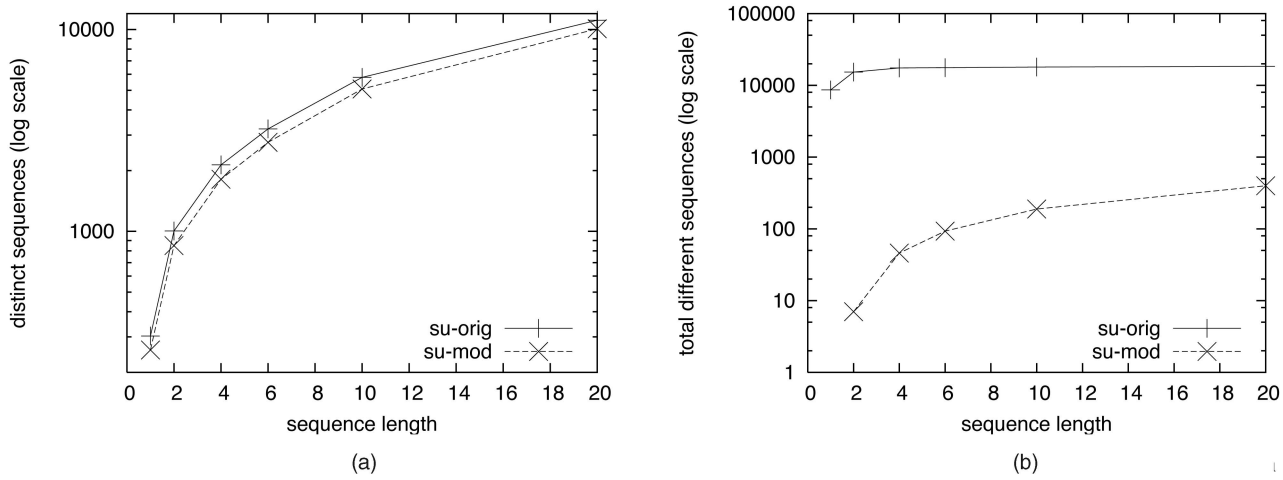


Fig. 1. Unique and different numbers of function call sequences in the original version of su and the version with `pam_authenticate` removed. (a) Number of distinct call sequences in the original and modified versions. (b) Function call sequences present only in a single version.

different between the two executions, but little more can be learned. Therefore, we compared the data between the two more directly. Fig. 1b shows plots of the number of calls appearing only in one version of `su` and not in the other (using a logarithmic scale). Again, as k grows, the number of sequences appearing in the original version and not in the modified one quickly becomes too large for a human to analyze easily (unless obvious patterns are present), but the number of sequences appearing in the modified version and not the original stays easily viewable until a sequence length of about 6 is used. We chose a length of 4. Using the instance-based method described earlier, a comparison between sequences in the original and modified versions of `su` shows

$k = 4$	different sequences
only in `su`-original	17497 (370 distinct)
only in `su`-modified	46 (all 46 distinct)

Of the 370 distinct sequences appearing only in the original version, we also learned that 14 sequences of length 4 were called with unusually high frequency (see Table 1).

These 14 sequences in Table 1 represent nearly half the entire program execution. Looking more closely at the 14 sequences (which is easy to do, since they stand out so voluminously from the others), all but one are clearly related to MD5, which does checksumming and encryption. This is an obvious clue to a forensic analyst that authentication is involved in the anomaly.

By comparison, it would not have been obvious what functionality was removed had we looked only at kernel calls, because they tend to be more utilitarian and less descriptive. For example, setting $k = 1$ (note that a sequence of length $k = 1$ is the same thing as a single call) and looking only at kernel calls, the calls absent in the modified version of `su` but present in the original were `setitimer` and `write`. Given $k = 2$, the number of sequences present in one and not the other jumps considerably. Among the sequences were the calls `fstat`, `ioctl`, `nosys`, `sigaction`, `getuid`, and `stat`. These are clearly useful pieces of information but not as descriptive as function calls.

Might there be an even easier way of finding the anomalous function call sequences? When $k = 2$, the results change significantly:

$k = 2$	different distinct sequences
only in `su`-original	161
only in `su`-modified	7

The reduced number of anomalous sequences makes the data much easier to look through. Using $k = 1$,

$k = 1$	different distinct sequences
only in `su`-original	45
only in `su`-modified	0

In fact, we can summarize the relevant calls for $k = 1$ in four lines:

$k = 1$ sequence	# total occurrences	% of total program
MD5Update	5538	14.91%
MD5Final	1002	2.70%
MD5Init	1002	2.70%
MD5Pad	1002	2.70%
Total	9547	18.69%

In this experiment, $k = 1$ provides a list of differing function calls that provided large clues about what was anomalous in the program. This is not always the case. In experiments with other programs, we discovered that $k = 1$ showed no differences at all, and a minimum value of $k = 2$ or even $k = 4$ was needed. Likewise, we discovered that $k = 4$ provided manageable results similar to those in this experiment, but $k > 4$ provided too many. In describing future experiments, we will choose a value of k that shows a differing number of sequences for at least one of the code versions to be greater than 1 and less than 20. In most cases, that means either $k = 2$ or $k = 4$.

The experiment also showed that function calls provide an additional value beyond that provided by system calls.

TABLE 1
Sequences of Length 4 Appearing Only in the Original Version of `su` and Not the Version with the Call to `pam_authenticate` Removed

$k = 4$ sequences	# total occurrences	% of total program
MD5Update,memcpy,MD5Update,memcpy	3533	9.51%
memcpy,MD5Update,memcpy,MD5Update	1528	4.11%
MD5Final,MD5Pad,MD5Update,memcpy	1002	2.70%
memcpy,MD5Update,memcpy,memcpy	1002	2.70%
memcpy,memcpy,memcpy,memcpy	1002	2.70%
MD5Update,memcpy,memcpy,memcpy	1002	2.70%
MD5Init,MD5Update,memcpy,MD5Update	1002	2.70%
MD5Pad,MD5Update,memcpy,MD5Update	1002	2.70%
memcpy,MD5Update,memcpy,MD5Final	1001	2.70%
memcpy,MD5Final,MD5Pad,MD5Update	1001	2.70%
MD5Update,memcpy,MD5Final,MD5Pad	1001	2.70%
memcpy,memcpy,memset,MD5Init	1000	2.69%
memcpy,memcpy,MD5Init,MD5Update	1000	2.69%
memcpy,MD5Init,MD5Update,memcpy	1000	2.69%
Total	17076	46.0%

Using $k = 2$ and looking only at system calls, there are 23 sequences of system calls (19 distinct) occurring in the original version of `su` and not the modified. The most significant differences are as follows:

$k = 2$ sequences of system calls only in `su`-orig	# total occurrences
ioctl, write	2 (0.27%)
nosys, fstat	2 (0.27%)
getuid, stat	2 (0.27%)
open, close	2 (0.27%)
lstat, open	2 (0.27%)

These sequences suggest that an anomaly is occurring but do not describe what the anomaly is. Indeed, none of the sequences would provide any indication to most forensic analysts as to where to look in the source for the anomalous behavior. Contrast this to the much more useful perspective that the sequences of function calls provided. Also, we can see that though the amount of data captured by recording function calls rather than system calls alone is 20-200 times higher, the amount of data necessary for an analyst to examine is not nearly as high. In this case, the number of distinct different function call sequences is only seven times higher than the number of distinct different system call sequences, with some sequences appearing so frequently that they immediately stand out. The function call data is more useful and does not require much more work to examine.

4.1.2 su Experiment 2

We performed a second experiment with `su`, where we modified `su` to run `pam_authenticate` but ignore the results rather than just removing the function entirely. In Fig. 2, we show a number of sequences appearing in one version but not the other. Again, we see that a sequence of length 4 gives a manageable amount of results for sequences appearing only in the original version of su, with four of the 13 sequences being

$k = 4$ sequences in `su`-original, not in `su`-modified
strcmp,pam_set_item,memset,free
crypt_to64,crypt_to64,strcmp,pam_set_item
crypt_to64,strcmp,pam_set_item,memset
sys_wait4,login_getcapnum,cgetstr,cgetcap

That said, $k = 2$ still gives us all we need to investigate the anomaly. For a sequence of length 2, there are 13 distinct sequences occurring in `su-original` and not in `su-modified`. One is "`strcmp`, `pam_set_item`," which is sufficient to raise concerns in any forensic analyst's mind because it indicates that the result of the authentication is not being set (`pam_set_item`) after the check (`strcmp`).

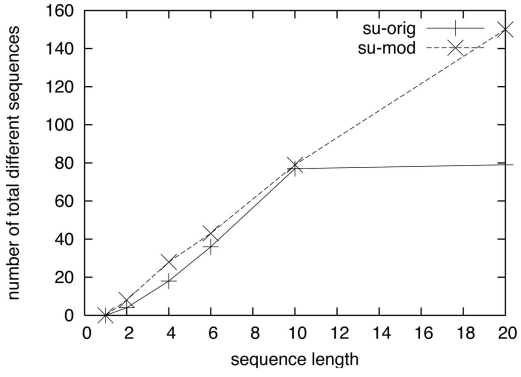


Fig. 2. Number of function call sequences appearing only in the original version of `su` and the version modified to ignore the results of `pam_authenticate`.

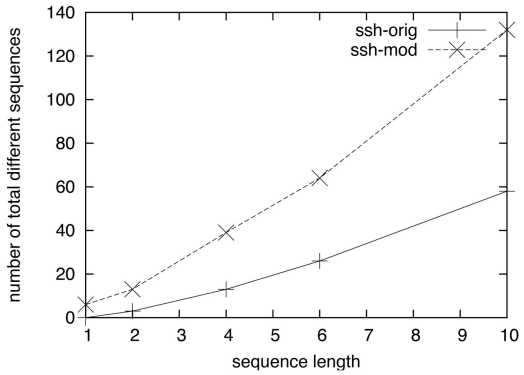


Fig. 3. Number of function call sequences appearing only in the original version of `ssh` and the version modified to echo the password back.

By comparison, looking only at system call traces, results are again less forensically useful because they are less descriptive. There are no different sequences of system calls with $k = 1$ or $k = 2$. With $k = 4$, we see three system calls (three distinct) in the original version and not in the modified and four system calls (four distinct) in the modified version and not in the original. Unlike function calls indicating a relationship with authentication and cryptographic routines, however, we instead see

syscall sequences only in `su`-modified
ioctl,close,close,sigaction
setpgid,ioctl,close,close
close,close,sigaction,sigaction

syscall sequences only in `su`-original
ioctl,close,sigaction,close
sigaction,close,sigaction,sigaction
setpgid,ioctl,close,sigaction
close,sigaction,close,sigaction

The above table indicates something suspicious involving a socket call, leading to an inference that there is a problem involving interprocess communication. There is nothing indicating the nature of the problem. This again emphasizes the value of function call sequences, which clearly shows an authentication issue. In what follows, we shall focus only on function call sequences rather than the differences between function call sequences and system call sequences.

4.2 Spyware

4.2.1 ssh Experiment 1

`ssh` is key to accessing systems over a network. Hence, it offers opportunities for malice, especially when the attacker has access to a password, private key, or other authentication token. Consider two versions of the `ssh` client: the original and one that is modified to do nothing more than echo the password back to the terminal. We wish to determine which sequence length will alert the analyst to the change. Fig. 3 shows the number of sequences that exist only in the executions of the original and modified versions of the `ssh` client. When a human analyst looks at a list of function call sequences flagged as anomalous, he or she can most easily spot the differences when at least one list of sequences appearing in one execution but not the other is relatively short. In this case, sequence lengths of k larger than 2 or 4 can cause the resulting sequences to become unmanageable. To the degree that an anomaly can be caught by sequences of length 2 or 4, an analyst should use those smaller values. It is difficult for an analyst to determine whether he or she has captured an anomaly with a sequence length of 2 or 4, but if an anomaly exists at all, a larger value of k is probably unnecessary.

In this case, the comparison between each version with sequence length $k = 4$ results in the following:

$k = 4$	total different sequences
only in `ssh`-original	13 (13 distinct)
only in `ssh`-modified	39 (39 distinct)

Many of the functions in the anomalous sequences have obscure names: `rlock_release`, `rtld_bind`, `rtld_bind_start`, and `localeconv`. Although the sequences clearly differ in the executions of both programs, the interpretation of the differences is not clear. Therefore, we include the function *returns* as tokens in addition to function call points. In what follows, the function returns are indicated with the suffix `-RET`:

$k = 2$	total different sequences, including function returns
only in `ssh`-original	6 (3 distinct)
only in `ssh`-modified	25 (25 distinct)

Now, using the additional data along with our previous methods, we discover the following variances in between all of the somewhat obscure sequences. In them, one function stands out immediately: `read_passphrase`. Table 2 shows sample sequences that appear only in each version of `ssh`.

These sequences in Table 2 indicate a location that needs to be examined more closely. Knowing about the `read_passphrase` call directs an analyst's attention to the routine in the source code, simplifying the investigation.

We mentioned earlier that shorter sequences would potentially be even easier to analyze if they catch the anomaly. As it turns out, $k = 2$ does catch the anomaly. In the two distinct sequences in the original version, the `read_passphrase` call stands out. In fact, `read_passphrase` is one of the only three nonkernel calls in the list, which further indicates what we get by looking at all calls rather than just kernel calls. In the 25 distinct sequences in the modified version, we show a selection of the sequences in the following chart. All but three involve string operations, which would help a forensic analyst find the offending location in the source code:

$k = 2$ sequences in `ssh-original`
memset-RET,read_passphrase-RET
BN_copy-RET,BN_div-RET
BN_div,BN_copy

TABLE 2
Sequences of Length 4 Only in a Single Version of ssh

$k = 4$ sequences in ssh-original, including function returns
memset-RET,read_passphrase-RET,input_userauth_info_req,packet_put_cstring
read_passphrase,memset-RET,read_passphrase-RET,input_userauth_info_req
xstrdup-RET,read_passphrase,memset-RET,read_passphrase-RET

$k = 4$ sequences in ssh-modified, including function returns
fprintf-RET,read_passphrase-RET,input_userauth_info_req,packet_put_cstring
vfprintf-RET,fprintf-RET,read_passphrase-RET,input_userauth_info_req
memset-RET,read_passphrase,rtld_bind_start,rtld_bind

$k = 2$ sequences in ssh-modified
fprintf-RET,read_passphrase-RET
vfprintf-RET,fprintf-RET
vfprintf,vfprintf

The inclusion of function returns effectively doubles the amount of data collected. However, as with the addition of function calls to system calls, the increase in the amount of data collected does not necessarily result in a proportional increase in the amount of data a human needs to examine. In our experiments, when the amount of data collected doubled, the increase in the amount of data presented to the analyst was often closer to 50 percent.

Nevertheless, the resulting increase in the data to collect and analyze was the reason why we do not use function returns in all of our experiments. There are some cases where function returns are helpful and others where they are unnecessary. As with the value of the sequence length parameter, an analyst will need to decide whether or not function returns should be included.

4.2.2 ssh Experiment 2

This example recreates a malicious program that was used by an intruder at the San Diego Supercomputer Center in 2004 and 2005 [51]. In the program, the intruder modified the ssh client program to capture user passwords and send the passwords via a network socket to another machine, which collected them. The difference in the number of function call sequences in the original and modified versions of ssh are shown in Fig. 4. The initial comparison shows the following:

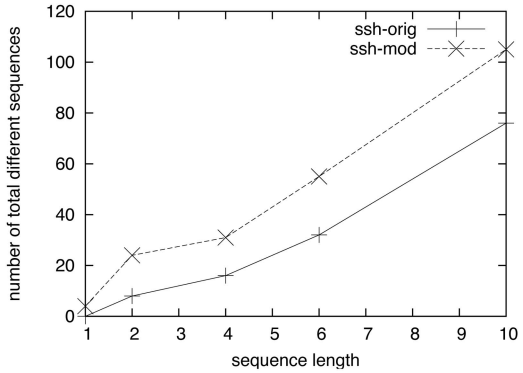


Fig. 4. Difference in the number of function call sequences in the original version of ssh and the version modified to send the captured password over a network socket.

$k = 2$	total different sequences
only in ssh-original	8 (4 distinct)
only in ssh-modified	24 (12 distinct)

Several calls do stand out: read_passphrase, inet_aton, inet_addr, socket, and sendto.

$k = 2$ key sequences in ssh-modified
inet_aton-RET,inet_addr-RET
read_passphrase,sys_close-RET
sys_close-RET,read_passphrase-RET
inet_addr,inet_aton
inet_addr-RET,read_passphrase
socket-RET,read_passphrase
sendto-RET,read_passphrase

In these examples, the four network-related functions, inet_aton, inet_addr, socket, and sendto (which is used to send data), might capture an analyst's attention. The first two are libc calls, and the last two are system calls. Whereas anomaly detection could have indicated a potential anomaly in the program, based upon the system calls, these forensic techniques draw an analyst's attention to the right place in the code to *understand* the anomaly. In this case, the libc calls help flag the anomaly, and using that information, further exploration in the code by a forensic analyst would reveal the purpose of the calls.

One complication results from the way Pin collects function calls and function returns. Due to technical limitations, Pin occasionally misses less than 1 percent of the calls made. For example, note that in the above set of key sequences, sendto-RET appears (showing the return from the sendto function), but the sendto call itself does not appear. Given the low volume of these missed calls, the results of the analysis are not affected.

4.3 Ignoring Permissions in the File System

In this example, we have recreated an experiment performed previously using anomaly detection techniques [3], [20], but in our experiment, we have used function calls instead of just system calls. The experiment involves a bug in an older version of lpr that allows an attacker to create or overwrite any file on the system [52]. In this case, the bug was exploited to replace /etc/passwd with /tmp/passwd.

Using $k = 2$ since that has been successful so far, we see 48 sequences (38 distinct) in the "test" version of lpr that are not in the "safe" corpus. In Table 3, the function calls

TABLE 3
Selected Sequences of Length 2 Appearing Only in `lpr-Modified`

selected $k = 2$ sequences only in `lpr-modified`	# occur
close, open	1
close, close	2
seteuid,creat	1
seteuid,sys_read	2
swhatbuf,sys_fstat	1
sys_write,**nfile**	1
copy,card	1
nfile,sys_umask	1
swrite,swrite	1
swrite,rtld_bind_start	1
printf,rtld_bind_start	1
swsetup,smakebuf	1
open,**copy**	1
link,error_unthreaded	1
sys_unlink,error_unthreaded	2
sys_unlink,sys_unlink	2
sys_unlink,rtld_bind_start	1
vfprintf,vfprintf	1
smakebuf,swhatbuf	1
ioctl,sfvwrite	1
sys_umask,fchown	1
sys_read,sys_write	2
cleanup,signal	1
malloc,isatty	1
sys_sigaction,seteuid	1
error_unthreaded,sys_unlink	2

that are part of `lpr` are in boldface font, and the system calls and `libc` calls are in standard font.

When one is printing a file, one may be surprised to find that files are created, copied, and removed and have their permissions changed in the process. Though each of the sequences in the table above appear only once or twice, we can see repeated calls to `sys_unlink` and `seteuid`, which stand out among these results as being important and suspicious. What actually helps to find the bug, however, is the `copy` function call in line 7 of the Table 3, which is a routine in `lpr.c` that creates a file and copies a file descriptor.

The `copy` function contains a call to `creat`. `creat` is a legitimate system call, though it is now deprecated in FreeBSD. An analyst who knows of the deprecation might be suspicious of a call to `creat`, because unlike `open`, it is now known to be one of the two steps in a nonatomic method for creating and opening a file, which must be used carefully to avoid race conditions. At the time the exploit was released, an analyst would not necessarily have known that `creat` might be a problem, however.

The `sys_unlink` and `seteuid` calls that we see in the table are not themselves the vulnerability but are the effect of the exploit, since the `creat` call allows files to be arbitrarily overwritten (`sys_unlink`) and permissions reset (`seteuid`).

4.4 Buffer Overflows

In the first three experiments, we have shown full real examples of actual exploits for the purpose of demonstrating the complete process of using our methods. However, the results are also partially demonstrating the skill of the analyst, not the simple results of our method. Therefore, in the following experiments, we use simplified examples rather than actual attacks. This is partially because the simple examples show the essence of the attack and analysis, and embedding them in more complex exploits simply requires that the analyst winnow them out. It is also due to space limitations, as there is not enough room for a detailed analysis of two more large attacks that use these exploits. Therefore, we present examples of the two attack approaches without the extraneous entries.

There are many examples of attacks that exploit buffer overflow vulnerabilities. One can invoke a `root` shell directly, copy a setuid `root` version of a shell into `/tmp`, or, instead of working directly to gain arbitrary access, write *shellcode*[5] to perform exactly the desired operations directly. Unfortunately, at this point, we have not discovered a universal pattern to buffer overflow exploits and therefore cannot detect one directly. What we can do at this point is understand when the order of events is unusual. Consider the following piece of code [53]:

```
int main(int argc, char *argv[]) {
    char buffer[500];
    strcpy(buffer, argv[1]);
    printf("Safe prog?"); return 0;
}
```

We run this program, called `vulnerable`, in two different ways. The first way is one that is "safe" and does not exploit the vulnerability by overflowing the buffer. It simply passes 499 instances of the ASCII character "a" as the argument to `vulnerable`. The second way exploits the vulnerability. It copies more than 500 characters into the buffer. The characters include *shellcode* that returns a shell owned by `root`.

Although we see a discrepancy between the sequences for the safe and exploited programs, the reason for the difference, at first, is unclear. As we did in a previous example, we add indications of where the functions return. The following variances appear:

$k = 4$ sequences only in non-overflowed `vulnerable`
vfprintf-RE, printf-RET, main-RET, start
printf-RET, main-RET, start, rtld_bind_start
main-RET, start, rtld_bind_start, rtld_bind

The exploited program contains a sequence of length 3: `vprintf-RET`, `printf-RET`, `main-RET`. The nonexploited version of the program continues after `main` ends with additional functions and finally ends with the `exit` syscall. The exploited version stops at the end of `main`. A forensic analyst who saw these results would realize that `exit` is being skipped in one version (the exploited version), most likely because the program has been altered. This means that all the cleanup functions `exit` invokes are also skipped.

Our difficulty in analyzing buffer overflows stems from a limit of the dynamic instrumentation tool we used. Pin currently allows analysts to instrument a code written to the stack and then executed, but a special command must be called between the time the code is written and the time it is

5. Shellcode refers to the assembly language instructions in a buffer overflow exploit designed to provide a shell.

executed. Unfortunately, implementing this special command would distort our view of the buffer overflow. That said, the evidence of the buffer overflow already exists in our view, even if the exact nature of the overflow does not. The effects of some exploits will be more apparent than the exploit itself. In this case, the exploited program does not exit (the effect) because it is actually spawning a program (the exploit itself, which we cannot see but can infer).

4.5 Trojan Horses

In an early version of Unix, there was a flaw in `vi`[6] such that when an editing session was terminated unexpectedly, it would mail the user about how to recover the data from the lost session by calling the mail program without a full path (for example, `/bin/mail` but with simpler a program name: `mail`). As a result, if a binary called `mail` were placed earlier in the search path order than `/bin`, the binary, which could be a Trojan horse, would be executed. Further, since `vi` used a privileged program to store temporary files on the system and that program called `mail`, the Trojaned version of `mail` would be executed with superuser privileges.

Our experiment involves a very minimal program:

```
int main() {
    FILE * F = popen("date","r");
    printf("Opened\n"); pclose(F);
}
```

In this program, the `date` program is called with `popen`, just as the original bug in `vi`. The circumstances are such that the path is set to

```
.:/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/
sbin:/usr/local/bin
```

The system's `date` program is located in `/bin`. In the first case, that is the only `date` in the path. In the second case, a "rogue" `date` program is also in the current working directory. In reality, it could be easier to modify a user's path without their knowledge or to put a malicious program higher in their "search path" than the system's than to obtain `root` access. This is because inserting a malicious program only requires access to a *user's* account, not the `root` account. Therefore, any number of password-sniffing social engineering or other techniques might enable an attacker to do this.

We test the same program under both circumstances. Differences in each case show up with values of $k \geq 6$. With $k = 6$ and comparing the library calls of the program that is being "tricked" into executing the Trojan, not the Trojan itself, the results show the following:

$k = 6$ sequences only in version calling Trojan
malloc,strncpy,xstat64,malloc,xstat64,access
strncpy,xstat64,malloc,xstat64,access,free
xstat64,malloc,xstat64,access,free,xstat64

Without even looking at the function parameters themselves, in the first execution, we can see multiple calls relating to `malloc`, `strncpy`, `access`, `free`, and `xstat64`.

```
sean:x4nPA20z:500:500:Sean the Wizard:/home/sean:/bin/sh
sean2::0:0::/home/sean:/bin/tcsh
```

Fig. 5. Two different possible entries in `/etc/passwd`.

Looking more carefully at the raw data of function calls, parameters, and return values (the latter two are available as a result of using `ltrace` in this experiment rather than *Pin*), we can see why the version that runs the system `date` does an `xstat64` (related to `stat`) on the current working directory and `/sbin` before `/bin`. Further, the first two return nonzero values, which indicates the nonexistence of a file, whereas the `xstat64` returns zero in the `/bin` directory for `date`. Conversely, when the Trojaned `date` is in the directory, there is only one `xstat64`, and the program goes no further.

Intuitively, we know why this is the case. The version of the test program that gets tricked into calling the Trojan searches its path (and, hence, uses `xstat64`) in different sequences than the version that calls the system `date`. The results are misleading, given that there are actually *more* total sequences in the version calling the system `date` than the Trojan but more *distinct* sequences in the other version. This is an anomaly that sometimes occurs. In this case, it occurs with $k < 14$. Nevertheless, the experiment shows how sequences of function calls are helpful in analyzing Trojan horses.

The results of this experiment show not only the value of tracing library calls across forks, but also the value of and ability to analyze call parameters and return values.

4.6 Invalid Parameters

Another early Unix flaw occurred in `chsh` and `chfn`. Those programs would accept a new shell or full username from the input line, even if the input contained a *colon* character.[7] Since the password would get written into the `/etc/passwd` file with the colons intact, a `uid` or `gid` of zero, indicating superuser privileges (or any arbitrary value), could be appended after a colon and overwrite the existing values. For example, the first line in Fig. 5 could be changed to the second line.

The solution is either to have the program stop reading input from the user when it sees either a colon *or* a newline or to have the program simply stop writing output to the `passwd` file when it sees a colon or newline.

A modern example of this flaw is the conflict between Unix and Mac OS X, wherein Unix uses a forward slash ("/") to designate directories, whereas in Mac OS X, a file can be written with a forward slash contained in the name of a file or directory.

We do not implement a method for forensic analysis of this flaw because it requires observing only the call parameters and not the calls themselves and, therefore, our exact techniques are not applicable. Instead, the experiment would benefit from more sophisticated data mining techniques. However, the method itself is straightforward: Capture all calls and parameters and use data mining techniques to build a "normal" database of the parameters given to the `chsh` command. The results should include histograms of character frequencies given as input. We can assume that there will be large amounts of alphanumeric characters and even many symbolic characters. Some characters (such as control characters) are

6. See slide 171 in [54, p. 90].

7. See slide 85 in [54, p. 47].

disallowed and will not appear at all. If a colon character were to appear, even an automated detection system could know immediately that something unusual had been input and should be flagged for observation.

5 Future Work

We now describe work that could be done in the future to enhance or augment our methods.

5.1 HMMs

Generating HMMs of function calls requires considerable computing resources, because the complexity is $O(n^2)$, where n is the number of calls in a trace. Using a supercomputer to generate these would speed up the task considerably, which could make using HMMs practical in limited situations. These models of function calls should be generated so that the results could be compared to the results described in this paper.

5.2 Data Mining

Machine learning techniques should also be revisited. Function calls present more *features* than other data, which allows for higher accuracy. For example, in addition to the function calls, users, programs, and pathnames, there are also return values, arguments, and their types. The *k-nearest-neighbor* classification algorithm that was previously used in intrusion detection research is one possible classification algorithm that could be used to process the data. Another method of detecting possible anomalies is to measure the *entropy* of the argument and return data to determine whether the value appears normal or somehow anomalous.

Data mining techniques may report many anomalies that are not attacks. Analyzing function call sequences may produce models that constrain and focus the data mining techniques to minimize those false positives. The next step is to model the function call data and use automated classification techniques to search for previously unknown anomalies. This will require many experiments from a wide variety of program executions (both attacking and normal). From these experiments, enough data will be gathered to test these techniques.

5.3 Sessions

One method of avoiding detection of existing anomaly detection and forensics software is to have programs work together to exploit a vulnerability such that each program individually is doing something seemingly normal, but together could be exploiting a vulnerability. Therefore, the techniques presented in this paper should be applied to not only anomalous programs but also anomalous *sessions* and *users* and, therefore, also *masquerade attacks* [55] and *insiders* [56].

5.4 Masquerade Attacks

One technique to attempt to defeat our methods is mimicry attacks [57], [58]. Mimicry attacks are an attempt to make an attack on a computer system seem like another attack or perhaps seem like normal computer use and not an attack at all. They are an attempt at misdirection and confusion. Mimicry attacks are a concern with any kind of intrusion detection or forensic analysis. However, the fact that function calls encompass system calls helps significantly in this regard. If function parameters and return values to system calls (and system library calls) were recorded, the execution trace would show whether the calls were truly mimicry (for example, no-ops) or an actual execution. Further, many of the programs that are exploited to attack a system locally are not user-compiled programs at all but are rather already compiled and on the system, because they are quite often setuid `root`. Therefore, mimicry attacks would generally not apply in this situation since the setuid `root` program is almost always unwritable by non`root` users, and also using the setuid `root` program to execute a *new* binary would be significantly easier than adding functions and instructions to the exploited binary for mimicry purposes. In any case, in this situation, function call traces would be no more susceptible to mimicry than system call traces. To provide a greater assurance of avoiding mimicry attacks, the technique of interleaving known (but innocuous) calls with actual calls [59] could address the issue for both system calls and function calls, albeit at a considerable computational cost. More research might reveal ways in which masquerade attacks could be dealt with even more effectively in forensic analysis.

5.5 Implementation

Precisely measuring the effectiveness of forensic methods is hard. Even experiments that claim to objectively and thoroughly evaluate intrusion detection systems are incomplete, since they rely on a corpus of "good" and "bad" tests, yet none of these can practically do a complete evaluation of all possible experiments.

We intend to implement these forensic tools and techniques on systems that receive broader use. Ultimately, it is the use of these tools and techniques and the feedback from them that will help to both evaluate its effectiveness and refine it to improve utility and efficiency.

6 Practical Issues

If this system were to be implemented and used on a live system, there are a number of considerations that would enhance its practicality. We will briefly discuss a few of them.

From an instrumentation standpoint, combining the use of a dynamic instrumentation tool and a static binary rewriter could help performance. The kernel and related system binaries could be instrumented statically using the binary rewriter, and all user binaries could be forced to be run through a dynamic instrumentation tool by modifying the `exec` system call. Ideally, a tool that had the combined benefit of capturing *all* calls, as does Pin, as well as all parameters and return values, and the ability to follow forks and vforks, as does the `ltrace` tool, and finally captures calls made by a self-modifying code would be ideal. At the time of this writing, the authors are not aware of any such tool.

Because Pin runs in the user space, it might be possible to bypass reliable function call recording. However, when run as `root`, Pin is no more vulnerable than the kernel. Further, since all programs are instrumented, the act and methods of bypassing Pin would be recorded. Finally, the absence of data from Pin or the intentional distortion of data would be observed and raise a red flag. A variety of techniques have emerged to defend against attacks on system call tracing mechanisms, which could also be implemented [60], [61], [62].

A method of storing audit logs in a tamper-resistant way would also be helpful. Possible tamper-resistant solutions include encryption [63], streaming real-time logs to an independent system [8], [64] in an entirely different security domain, and/or a write-once mechanism.

The forensic analysis might involve a detailed examination of the source code after an analyst notices a function call among the anomalous sequences that is descriptive and significant. The source code for the operating system and related binaries should be kept available for this purpose. However, if users are allowed to compile their own binaries, a good practice would be to put the source code in escrow. A policy disallowing binaries not compiled on the instrumented system could be enforced using features such as NetBSD's *verified-exec* [65] so that the source code is always captured for later analysis.

7 Relationship of Forensics and Intrusion Detection

Auditing for intrusion detection began as largely a manual process [66]. Since that time, the field that began as intrusion detection has divided into a field that has focused on an automated process [19] and one that has focused on forensic analysis. The problems of forensic analysis and intrusion detection overlap. Both are based on system behavior, and at a high level, both are looking for something that seems wrong. The differences between the problems, however, are significant.

The primary goal of intrusion detection is to determine whether an intrusion has occurred [1]. Intrusion detection is ideally automated, online, and real time. The goal is to detect an intrusion as quickly as possible so that the damage can be contained. The response to a suspected intrusion can be either manual or automated [22], but that is distinct from detection [1], [14]. Forensic analysis is one possible *postmortem* response to a suspected intrusion.

In the forensic process, currently a human—the analyst—must often interpret the results of unpredictable activity. One reason for this is that the state of the art of forensic analysis research is significantly behind intrusion detection. The process of analyzing and understanding unexpected events—those that are not predefined in an attack or event model [67]—is currently hard, whereas the process of detecting unexpected events in the intrusion detection process is easy. Another reason is that in some cases, forensic analysis ultimately needs to account for events that occur outside the computer system. If an intruder logs into a system using a password, an important question is how it was obtained. It may have been the result of a password sniffer, or it may have been the result of eavesdropping on a conversation in a hallway.

Despite the overlap between the intrusion detection and forensic analysis processes, they are distinct problems. Because of this, there has not been as much cross fertilization as one might wish. In this paper, we have borrowed one technique from intrusion detection and, with modifications, successfully applied it to forensic analysis.

8 Conclusions

The set of all orderings of safe events on a computer system is finite (as a computer is a finite state machine), and upon many repeated executions, the total number of distinct sequences actually executed levels off asymptotically. If an ordering of events in that set occurs, the program is behaving anomalously. We cannot always determine this in real time, however, because nondeterministic events such as user inputs and variable system state cause programs to perform deterministic events at different times. However, when known "good" events do happen, they do so in an order that can often be predicted by gathering enough data about the safe operation of a program ahead of time. Therefore, we need to use anomaly detection techniques on data in a *postmortem* manner, even though this only tells an analyst whether the program is anomalous after the data for the entire program has been analyzed.

Anomalies among short sequences of function calls indicate not only that a *process* is anomalous but also the *part* of the process (code) that is anomalous. This significantly reduces the amount of the recorded function call data that actually needs to be analyzed by a human. We can then check the source code. A fundamental requirement of forensics is to understand not only *if* an anomaly is occurring but also exactly *where* it occurs. This requires detail without overwhelming volumes of data. Anomalous sequences of function calls fit this criteria better than other levels of abstraction and analysis methods previously used. Function calls are also a helpful abstraction, because humans generally find them more descriptive than system calls alone. Although we do not claim that function calls, rather than system calls alone, are *required* to analyze all attacks, we have shown that function calls can help to analyze the attacks much more easily due to the descriptiveness of the calls. The function call sequence length, previously shown to have no optimal answer, appears to be most desirable when $1 \leq k \leq 10$, because it produces manageable but not overwhelming amounts of data.

That a program execution has a finite set of function call sequences suggests that it should be possible to model programs formally as sequences of function calls. Correlating the types of anomalies that an analyst looks for generates automated common models of anomalous behavior based on those models. At present, we do not know enough about the behavior of most programs to formulate such a formal specification or some other formal model that would define the vulnerabilities better [68]. Given the success of our experiments, the development of more universal analysis techniques and formal models that show commonalities between exploits seems promising.

> *"Is there any point to which you would wish to draw my attention?"*
>
> *"To the curious incident of the dog in the night-time."*
>
> *"The dog did nothing in the night-time."*
>
> *"That was the curious incident," remarked Sherlock Holmes.*
>
> —Sir Arthur Conan Doyle, "Silver Blaze,"
>
> *The Strand Magazine* (1892)

Anomaly detection techniques on function call data in the postmortem analysis allows one to find unexpected events, including the absence of expected function calls, as well as the presence of unexpected function calls. This enables an analyst to discover when an event that should occur does not.

Some techniques fail. Looking for calls that are "rare" in one version and "common" in the other did not bear much fruit in our experiments, although even those failed experiments suggest that the technique has potential value, and further experimentation is warranted. Additionally, more descriptive information should help address the previously unanalyzable scenarios. However, as we noted earlier, evaluation of these techniques is difficult and ultimately will require further study to determine which technique is most effective.

ACKNOWLEDGMENTS

This research was supported in part by a Lockheed-Martin Information Assurance Technology Focus Group 2005 University Grant, Award ANI-0330634, "Integrative Testing of Grid Software and Grid Environments," from the US National Science Foundation, and by the US Air Force Office of Scientific Research Multidisciplinary University Research Initiative (AFOSR MURI) Grant 41131-6865. Matt Bishop was supported by Awards CCR-0311671 and CCR-0311723 from the US National Science Foundation to the University of California, Davis. The authors wish to thank Robert S. Cohn and Steven Wallace at Intel for their enhancements to the FreeBSD version of Pin.

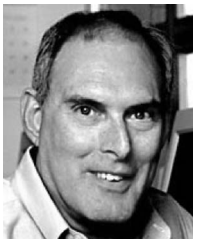
REFERENCES

[1] M. Bishop, *Computer Security: Art and Science.* Addison-Wesley Professional, 2003.
[2] S. Peisert, M. Bishop, S. Karin, and K. Marzullo, "Principles-Driven Forensic Analysis," *Proc. New Security Paradigms Workshop (NSPW '05),* pp. 85-93, Oct. 2005.
[3] S.A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion Detection Using Sequences of System Calls," *J. Computer Security,* vol. 6, pp. 151-180, 1999.
[4] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the Detection of Anomalous System Call Arguments," *Proc. European Symp. Research in Computer Security (ESORICS '03),* pp. 326-343, 2003.
[5] W. Lee and S.J. Stolfo, "Data Mining Approaches for Intrusion Detection," *Proc. Seventh Usenix Security Symp.,* pp. 26-29, Jan. 1998.
[6] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," *Proc. IEEE Symp. Security and Privacy (S&P '01),* pp. 144-155, 2001.
[7] A. Wespi, M. Dacier, and H. Debar, "Intrusion Detection Using Variable-Length Audit Trail Patterns," *Proc. Third Int'l Workshop Recent Advances in Intrusion Detection (RAID '00),* pp. 110-129, 2000.
[8] A. Goel, W. Feng, D. Maier, W. Feng, and J. Walpole, "Forensix: A Robust, High-Performance Reconstruction System," *Proc. 25th Int'l Conf. Distributed Computing Systems Workshop,* pp. 155-162, 2005.
[9] S.T. King and P.M. Chen, "Backtracking Intrusions," *ACM Trans. Computer Systems,* vol. 23, no. 1, pp. 51-76, Feb. 2005.
[10] D.S. Peterson, M. Bishop, and R. Pandey, "A Flexible Containment Mechanism for Executing Untrusted Code," *Proc. 11th Usenix Security Symp.,* pp. 207-225, Aug. 2002.
[11] N. Provos, "Improving Host Security with System Call Policies," *Proc. 12th Usenix Security Symp.,* pp. 257-272, 2003.
[12] S.-P. Shieh and V.D. Gligor, "Detecting Illicit Leakage of Information in Operating Systems," *J. Computer Security,* vol. 4, nos. 2-3, pp. 123-148, Jan. 1996.
[13] W. Osser and A. Noordergraaf, *Auditing in the Solaris Operating Environment.* Sun Microsystems, Inc., Feb. 2001.
[14] R.G. Bace, *Intrusion Detection.* Macmillan Technical Publishing, 2000.
[15] M. Bishop, "Profiling under Unix by Patching," *Software—Practice and Experience,* vol. 17, no. 10, pp. 729-740, Oct. 1987.
[16] H.H. Feng, J.T. Griffin, Y. Huang, S. Jha, W. Lee, and B.P. Miller, "Formalizing Sensitivity in Static Analysis for Intrusion Detection," *Proc. IEEE Symp. Security and Privacy (S&P '04),* 2004.
[17] J.T. Giffin, S. Jha, and B.P. Miller, "Efficient Context-Sensitive Intrusion Detection," *Proc. 11th Ann. Network and Distributed Systems Security Symp. (NDSS '04),* Feb. 2004.
[18] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous System Call Detection," *ACM Trans. Information and System Security,* vol. 9, no. 1, pp. 61-93, Feb. 2006.
[19] D.E. Denning, "An Intrusion-Detection Model," *IEEE Trans. Software Eng.,* vol. 13, no. 2, pp. 222-232, Feb. 1987.
[20] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, "A Sense of Self for Unix Processes," *Proc. IEEE Symp. Security and Privacy (S&P '96),* pp. 120-128, 1996.
[21] A.B. Somayaji, "Operating System Stability and Security through Process Homeostasis," PhD dissertation, Univ. of New Mexico, July 2002.
[22] A. Somayaji and S. Forrest, "Automated Response Using System Call Delays," *Proc. Ninth Usenix Security Symp.,* Aug. 2000.
[23] I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques,* second ed. Morgan Kaufmann, 2005.
[24] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," *Proc. IEEE Symp. Security and Privacy (S&P '99),* pp. 133-145, 1999.
[25] K.M. Tan and R.A. Maxion, "'Why 6?'—Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector," *Proc. IEEE Symp. Security and Privacy (S&P '02),* pp. 188-201, 2002.
[26] H. Inoue and S. Forrest, "Anomaly Intrusion Detection in Dynamic Execution Environments," *Proc. New Security Paradigms Workshop (NSPW '02),* pp. 52-60, 2002.
[27] L. Lankewicz and M. Benard, "Real-Time Anomaly Detection Using a Nonparametric Pattern Recognition Approach," *Proc. Seventh Ann. Computer Security Applications Conf.,* pp. 80-89, Dec. 1989.
[28] H.S. Vaccaro and G.E. Liepins, "Detection of Anomalous Computer Session Activity ("Wisdom and Sense")," *Proc. IEEE Symp. Security and Privacy (S&P '89),* pp. 280-289, 1989.
[29] J. Frank, "Artificial Intelligence and Intrusion Detection: Current and Future Directions," *Proc. 17th Nat'l Computer Security Conf.,* June 1994.
[30] 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," *Proc. Symp. Operating Systems Design and Implementation (OSDI '02),* 2002.
[31] M. Xu, R. Bodik, and M.D. Hill, "A 'Flight Data Recorder' for Enabling Full-System Multiprocessor Deterministic Replay," *Proc. 30th Ann. Int'l Symp. Computer Architecture (ISCA '03),* pp. 122-133, 2003.
[32] S. Narayanasamy, G. Pokam, and B. Calder, "BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging," *Proc. 32nd Int'l Symp. Computer Architecture (ISCA '05),* June 2005.
[33] M. Bishop, "A Standard Audit Trail Format," *Proc. 18th Nat'l Information Systems Security Conf.,* pp. 136-145, Oct. 1995.
[34] A.H. Gross, "Analyzing Computer Intrusions," PhD dissertation, Dept. Electrical and Computer Eng., Univ. of California, San Diego, 1997.
[35] S. Sitaraman and S. Venkatesan, "Forensic Analysis of File System Intrusions Using Improved Backtracking," *Proc. Third IEEE Int'l Workshop Information Assurance,* pp. 154-163, 2005.
[36] S.T. King, Z.M. Mao, D.G. Lucchetti, and P.M. Chen, "Enriching Intrusion Alerts through Multi-Host Causality," *Proc. 12th Ann. Network and Distributed System Security Symp. (NDSS '05),* 2005.
[37] P. Sommer, "Intrusion Detection Systems as Evidence," *Proc. First Int'l Workshop Recent Advances in Intrusion Detection (RAID '98),* 1998.
[38] P. Stephenson, "The Application of Intrusion Detection Systems in a Forensic Environment (extended abstract)," *Proc. Third Int'l Workshop Recent Advances in Intrusion Detection (RAID '00),* 2000.
[39] M. Bishop, "A Model of Security Monitoring," *Proc. Fifth Ann. Computer Security Applications Conf. (ACSAC '89),* pp. 46-52, Dec. 1989.
[40] F.B. Schneider, "Enforceable Security Policies," *ACM Trans. Information and System Security,* vol. 3, no. 1, pp. 30-50, Feb. 2000.
[41] K.W. Hamlen, G. Morrisett, and F.B. Schneider, "Computability Classes for Enforcement Mechanisms," *ACM Trans. Information and System Security,* vol. 28, no. 1, pp. 174-205, 2005.
[42] B.A. Kuperman, "A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources," PhD dissertation, Purdue Univ., 2004.

[43] E. Eskin, W. Lee, and S.J. Stolfo, "Modeling System Calls for Intrusion Detection with Dynamic Window Sizes," *Proc. DARPA Information Survivability Conf. and Exposition II (DISCEX II '01),* 2001.
[44] Aleph One, "Smashing the Stack for Fun and Profit," *Phrack,* vol. 7, no. 49, 1996.
[45] M. Bishop and M. Dilger, "Checking for Race Conditions in File Accesses," *Computing Systems,* vol. 9, no. 2, pp. 131-152, 1996.
[46] J.P. Anderson, "Computer Security Technology Planning Study," Technical Report ESD-TR-73-51, vol. II, ESD/AFSC, Hanscom AFB, Bedford, MA, Oct. 1972.
[47] R.P. Abbott, J. Chin, J.E. Donnelly, W. Konigsford, S. Tokubo, and D.A. Webb, "Security Analysis and Enhancements of Computer Operating Systems (RISOS)," technical report, Lawrence Livermore Laboratory, Apr. 1976.
[48] R. Bisbey and D. Hollingworth, "Protection Analysis: Final Report (PA)," technical report, Information Sciences Inst., May 1978.
[49] P. Neumann, "Computer Security Evaluation," *1978 Nat'l Computer Conf., AFIPS Conf. Proc.,* vol. 47, pp. 1087-1095, 1978.
[50] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI '05),* June 2005.
[51] A. Singer, "Tempting Fate," *;login:,* vol. 30, no. 1, pp. 27-30, Feb. 2005.
[52] 8LGM, "Unix `lpr` Security Advisory," http://www.8lgm.org/advisories/[8lgm]-Advisory-3.UNIX.lpr.19-Aug-1991.html, 2005.
[53] fides, "Simple Buffer-Overflow Exploits," *Collusion E-Zine,* vol. 23, May 2001, http://www.collusion.org/Article.cfm?ID=176.
[54] M. Bishop, "How Attackers Break Programs and How to Write Programs More Securely," *Proc. SANS 2002 Ann. Conf.,* May 2002, http://nob.cs.ucdavis.edu/bishop/secprog/sans2002.pdf.
[55] R.A. Maxion, "Masquerade Detection Using Enriched Command Lines," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '03),* pp. 5-14, 2003.
[56] M. Bishop, "The Insider Problem Revisited," *Proc. New Security Paradigms Workshop (NSPW '05),* pp. 75-76, Oct. 2005.
[57] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis," *Proc. IEEE Symp. Security and Privacy (S&P '01),* pp. 156-168, 2001.
[58] D. Wagner and P. Soto, "Mimicry Attacks on Host-Based Intrusion Detection Systems," *Proc. Ninth ACM Conf. Computer and Comm. Security,* 2002.
[59] H. Xu, W. Du, and S.J. Chapin, "Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths," *Proc. Seventh Int'l Symp. Recent Advances in Intrusion Detection (RAID '04),* pp. 21-38, 2004.
[60] T. Garfinkel, "Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools," *Proc. Network and Distributed System Security Symp. (NDSS '03),* 2003.
[61] T. Garfinkel, B. Pfaff, and M. Rosenblum, "Ostia: A Delegating Architecture for Secure System Call Interposition," *Proc. Network and Distributed System Security Symp. (NDSS '03),* 2003.
[62] M. Rajagopalan, M.A. Hiltunen, T. Jim, and R.D. Schilichting, "System Call Monitoring Using Authenticated System Calls," *IEEE Trans. Dependable and Secure Computing,* vol. 3, no. 3, pp. 216-228, July-Sept. 2006.
[63] B. Schneier and J. Kelsey, "Secure Audit Logs to Support Computer Forensics," *ACM Trans. Information and System Security,* vol. 2, no. 2, pp. 159-176, May 1999.
[64] D. Zagorodnov, K. Marzullo, L. Alvisi, and T.C. Bressoud, "Engineering Fault-Tolerant TCP/IP Servers Using FT-TCP," *Proc. Int'l Conf. Dependable Systems and Networks (DSN '03),* 2003.
[65] B. Lymn, "Verified Exec—Extending the Security Perimeter," *Proc. Linux.Conf.Au,* 2004.
[66] J.P. Anderson, "Computer Security Threat Monitoring and Surveillance," technical report, James P. Anderson, Fort Washington, PA, Apr. 1980.
[67] B. Schneier, "Attack Trees: Modeling Security Threats," *Dr. Dobb's J.,* vol. 24, no. 12, pp. 21-29, Dec. 1999.
[68] M. Bishop, "Vulnerabilities Analysis," *Proc. Second Int'l Workshop Recent Advances in Intrusion Detection (RAID '99),* pp. 125-136, Sept. 1999.

Sean Peisert received the PhD degree in computer science from the University of California, San Diego (UCSD) in 2007. He is currently a postdoctoral scholar in the Department of Computer Science and Engineering at UCSD. He works on issues of computer security, particularly forensic analysis and security policy modeling. He is a student member of the IEEE.

Matt Bishop received the PhD degree in computer science from Purdue University in 1984. He is a professor in the Department of Computer Science at the University of California at Davis, where he has been since 1993. He works on issues of computer security, especially on vulnerabilities analysis, intrusion detection, attack analysis, and data sanitization. He is a member of the IEEE.



Sidney Karin received the PhD degree in nuclear engineering from the University of Michigan in 1973. He is a professor in the Department of Computer Science and Engineering at the University of California, San Diego (UCSD), and is the former director of the San Diego Supercomputer Center (SDSC), which he founded in 1985. He is a member of the IEEE.

Keith Marzullo received the PhD degree in electrical engineering from Stanford University in 1984. He is a professor and the chair of the Computer Science and Engineering Department at the University of California, San Diego (UCSD), where he has been since 1993. He works on both theoretical and practical issues of fault-tolerant distributed computing in various domains.