On Gray-Box Program Tracking for Anomaly Detection

Debin Gao Michael K. Reiter Dawn Song
Carnegie Mellon University
dgao@ece.cmu.edu reiter@cmu.edu dawnsong@cmu.edu

Abstract

Many host-based anomaly detection systems monitor a process ostensibly running a known program by observing the system calls the process makes. Numerous improvements to the precision of this approach have been proposed, such as tracking system call sequences, and various "gray-box" extensions such as examining the program counter or return addresses on the stack when system calls are made. In this paper, we perform the first systematic study of a wide spectrum of such methods. We show that prior approaches can be organized along three axes, revealing new possibilities for system-call-based program tracking. Through an empirical analysis of this design space, we shed light on the benefits and costs of various points in the space and identify new regions that appear to outperform prior approaches. In separate contributions, we demonstrate novel mimicry attacks on a recent proposal using return addresses for system-call-based program tracking, and then suggest randomization techniques to make such attacks more difficult.

1 Introduction

A server program with buffer overflow or format string vulnerabilities might permit an attacker to commandeer a process running that program, effectively causing it to run the attacker's program, instead. In order to detect when this occurs, anomaly detectors have been proposed to monitor the system calls made by a process, in an effort to detect deviation from a known profile of system calls for the program it is ostensibly running. Such anomaly detectors have been proposed and used in many settings, including host-based intrusion detection systems (e.g., [4, 11, 19, 20]) and related sandboxing and confinement systems (e.g., [12, 22]).

Given the importance of system-call-based anomaly detection, numerous approaches have been proposed to improve their precision. Many of these approaches are seemingly orthogonal to one another, and while each has been demonstrated to improve precision (and often, increase cost) in isolation, how best to use these enhancements in combination is unclear. This is the primary question we address in this paper. In our analysis, we identify axes that are motivated by proposed enhancements and then empirically analyze the design space these axes define. Our analysis covers many regions not previously explored in prior work, including some that outperform previous approaches in our analysis. To our knowledge, this study is the first such systematic study of the design space for system-call-based anomaly detection.

As an initial study of this design space, we limit our attention to "gray-box" program monitoring techniques. In order to characterize whether a system call is anomalous, an anomaly detector builds a model of the normal system-call behavior of the program. We use "black box", "gray box" and "white box" to refer to the type of information the anomaly detector uses to build this model and to monitor the running process. Black-box detectors do not acquire any additional information other than the system call number and arguments that pass through the system call interface when system calls are made (e.g., [4, 17]). In contrast, white-box detectors examine all available information including the program being monitored, by statically analyzing (and potentially modifying) the source code or binary (e.g., [2, 5, 6, 20]). Gray-box approaches lie in between: the anomaly detector does not utilize static analysis of the program, but does extract additional runtime information from the process being monitored when a system call is made, e.g., by looking into the program's memory (e.g., [3, 16]). Here we focus on gray-box approaches (and a few black-box approaches as degenerate cases), again as an initial study, but also because white-box approaches are platform dependent and less universally applicable; see Section 2.

A consequence of limiting our attention to gray-box approaches is that any gray-box model of normal behavior depends on being trained with execution traces that contain all normal behaviors of the program. It is not our goal here to determine how to acquire adequate training data for a program. Rather, we simply assume we have adequate training data in our study; if this is not true, our techniques might yield false detections, i.e., they may detect anomalies that are not, in fact, intrusions.

In this context, this paper makes the following contributions:

1. We organize the design space of gray-box program tracking along three axes, that informally capture (i) the information extracted from the process on each system call; (ii) the granularity of the atomic units utilized in anomaly detection (single system calls or variable-length system call sequences); and (iii) the history of such atomic units remembered by the anomaly detector during monitoring. This framework enables us to categorize most previous approaches and to pinpoint new approaches that were not explored before.

2. We systematically study this design space and examine the cost and benefits of the various (including new) gray-box program tracking approaches. Exploiting richer information along each axis improves the detector accuracy but also induces additional costs, by increasing both the size of the model and the cost of gleaning additional information from the running process. Through systematic study, we compare the benefits (resilience against mimicry attacks) and costs (performance and storage overhead) of growing these parameters, and develop recommendations for setting them in practice. In a nutshell, our analysis suggests that by examining return addresses, grouping system calls into variable-length subsequences, and remembering a "window" of the two most recent program states permits an anomaly detector to track the program with good accuracy at reasonable runtime and storage overhead, and to prevent certain mimicry attacks that cannot be stopped in previous approaches.

3. We generalize prior work on mimicry attacks [18, 21] to demonstrate a previously unreported mimicry attack on systems that employ return address information as an input to anomaly detection. Specifically, prior work introducing the use of return address information largely disregarded the possibility that this information could be forged by the attacker.[1] While doing so is indeed nontrivial, we demonstrate how the attacker can forge this information. Despite this observation, we demonstrate that utilizing this information continues to have benefits in substantially increasing the attack code size. This, in turn, can render some vulnerabilities impossible to exploit, e.g., due to the limited buffer space within which an attacker can insert attack code.

4. Finally, we suggest how to use (white-box) randomization techniques to render the mimicry attacks mentioned above more challenging.

The rest of the paper is organized as follows. Section 2 introduces our proposed framework for gray-box program tracking, which covers most of the previous works in this area and our new proposals. Section 3 provides a detailed quantitative study of the space of gray-box program tracking. Section 4 presents our attack on a previously proposed anomaly detector to forge information and evade detection. In Section 5 we describe the randomization technique to make such attacks more difficult. Finally, we present our conclusion and future work in Section 6.

2 Framework for gray-box program tracking and new spaces

In system-call-based anomaly detection, the anomaly detector maintains state per process monitored, and upon receiving a system call from that process (and possibly deriving other information), updates this state or detects an anomaly. Similar to previous works (e.g., [16, 20]), we abstract this process as implementing a nondeterministic finite automaton $(\mathcal{Q}, \Sigma, \delta, q_0, q_\perp)$, where $\mathcal{Q}$ is a set of states including the initial state q_0 and a distinguished state $q_\perp$ indicating that an anomaly has been discovered; Σ is the space of inputs that can be received (or derived) from the running process; and $\delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is a transition relation. We reiterate that we define δ as a relation, with the meaning that if state $q \in \mathcal{Q}$ is active and the monitor receives input $\sigma \in \Sigma$, then subsequently all states q' such that $(q, \sigma, q') \in \delta$ are active. If the set of active states is empty, we treat this as a transition to the distinguished state $q_\perp$.

Below we describe how to instantiate $\mathcal{Q}$ and Σ along the three axes, thereby deriving a space of different approaches for gray-box program tracking. We further show that this space with three axes provides a unified framework for gray-box program tracking, which not only covers most of the previous relevant gray-box proposals, but also enables us to identify new ones.

1. The first axis is the runtime information the anomaly detector uses to check for anomalies. In black-box approaches, the runtime information that an anomaly detector uses is restricted to whatever information is passed through the system call interface, such as the system call number and arguments (though we do not consider arguments here). In a gray-box approach, the anomaly detector can look into the process's address space and collect runtime information, such as the program counter and the set of return addresses on the function call stack. Let S represent the set of system call numbers, P represent the set of possible program counter values, R represent the set of possible return addresses on the call stack. The runtime information an anomaly detector could use upon a system call could be S, $P \times S$, or $R^+ \times P \times S$ where $R^+ = \bigcup_{d \geq 1} R^d$.

The second and third axes are about how an anomaly detector remembers execution history in the time domain.

2. The second axis represents whether the atomic unit that the detector monitors is a single system call (and whatever information is extracted during that system call) or a variable-length sequence of system calls [23, 24] that, intuitively, should conform to a basic block of the monitored program. That is, in the latter case, system calls in an atomic unit always occur together in a fixed sequence.

3. The third axis represents the number of atomic units the anomaly detector remembers, in order to determine the next permissible atomic units.

The decomposition of execution history in the time domain into axes 2 and 3 matches program behavior well: an atomic unit ideally corresponds to a basic block in the program in which there is no branching; the sequence of atomic units an anomaly detector remembers captures the control flow and transitions among these basic blocks.

According to the three axes, we parameterize our automaton to represent different points in the space of gray-box program tracking. In particular, the set of states $\mathcal{Q}$ is defined as $\mathcal{Q} = \{q_0, q_\perp\} \cup \left(\bigcup_{1 \leq m \leq n} \Sigma^m\right)$,[2] and $\Sigma \in \{\mathbf{S}, \mathbf{P}, \mathbf{R}, \mathbf{S}^+, \mathbf{P}^+, \mathbf{R}^+\}$ where

$$\begin{array}{ll} \mathbf{S} = S & \mathbf{S}^+ = S^+ \\ \mathbf{P} = P \times S & \mathbf{P}^+ = (P \times S)^+ \\ \mathbf{R} = R^+ \times P \times S & \mathbf{R}^+ = (R^+ \times P \times S)^+ \end{array}$$

By this definition, the value of Σ captures two axes, including the runtime information acquired by the anomaly detector (axis 1) and the grouping of system call subsequences in forming an atomic unit (axis 2), while the value of n captures axis 3, i.e., the number of atomic units the anomaly detector remembers. Intuitively, growing each of these axes will make the automaton more sensitive to input sequences. (In fact, it can be proven that the language accepted by an automaton A_1 is a subset of the language accepted by automaton A_2, if A_1 has a "larger" value on axis 1 or axis 3 than A_2 and the same value as A_2 on the other two axes.)

Below we first describe how a variety of prior works fit into our unified framework:

- In one of the original works in monitoring system calls, Forrest et al. [4] implement (an anomaly detection system equivalent to) an automaton where $\Sigma = \mathbf{S}$ and $n \geq 1$ is a fixed parameter that was empirically chosen as $n = 5$. (For clarification on this choice, see [17].[3]) The transition function δ is trained by observing the sequence of system calls emitted by the program in a protected environment and on a variety of inputs. Specifically, if during training, the automaton is in state $q = (s_1, \ldots, s_m)$ and input s is received, then $(q, s, (s_1, \ldots, s_m, s))$ is added to δ if $m < n$ and $(q, s, (s_2, \ldots, s_m, s))$ is added otherwise.
- Sekar et al. [16] propose coupling the system call number with the program counter of the

process when the system call is made. (Sekar et al. modify the usual definition of the program counter, however, as described in Section 4.1.) That is, $\Sigma = \mathbf{P}$. This effort considered only $n = 1$. As in [4], the transition function is trained as follows: if during training, the automaton is in state q and input $\sigma \in \Sigma$ is received, then (q, σ, q') is added to δ where $q' = (\sigma)$.

- Feng et al. [3] propose additionally considering the call stack of the process when a system call is made. When a system call is made, all return addresses from the call stack are extracted; i.e., $\Sigma = \mathbf{R}$. Again, this work considered only $n = 1$. If during training, the automaton is in state q and input $\sigma \in \Sigma$ is received, then (q, σ, q') is added to δ where $q' = (\sigma)$.

- Wespi et al. [23, 24] suggest an anomaly detection approach in which training is used to identify a set of system call subsequences using a pattern discovery algorithm [13]. The result of the training is a set of variable-length system call sequences $\Sigma = \mathbf{S}^+$. They then define an anomaly detection system in which $n = 0$ (in our parlance); i.e., for each $\sigma \in \Sigma$, (q_0, σ, q_0) is added to δ.

Of the approaches above, only that of Wespi et al. [23, 24] utilizes nondeterminism (i.e., permits multiple active states simultaneously). All others above could be expressed using a (deterministic) transition function, instead.

Table 1 summarizes the prior work described above and identifies the new approaches we explore in this paper. We emphasize that this is not necessarily a complete list of prior work, and that we have not captured all aspects of these prior works but rather only those of interest here. To our knowledge, however, our analysis is the first that covers many of the regions in Table 1. Moreover, in certain regions that have received attention in prior work, the analysis has been incomplete. Notably, the analysis of Wespi et al. [23, 24] was performed on audit log records, not system calls, though they conjectured the technique could be applied to system call monitoring, as well. In such cases, our analysis here provides new insight into the effectiveness of these techniques when applied to system call monitoring.

Finally, we remind the reader that by restricting our analysis to approaches captured in the above model, we do not address various "white-box" approaches to system-call-based anomaly detection. Though we intend to incorporate these white-box approaches into our future analysis, our reason for precluding them from this initial study is that they are generally more platform sensitive or require stronger assumptions, and thus are generally less applicable than gray-box approaches. For example, some require source code (e.g., [20]) and those that do not are platform specific. Most notably, the complexity of performing static analysis on x86 binaries is well documented. This complexity stems from difficulties in code discovery and module discovery [14], with numerous contributing factors, including: variable instruction size;[4] hand-coded assembly routines, e.g., due to statically linked libraries, that may not follow familiar source-level conventions (e.g., that a function has a single entry point) or use recognizable compiler idioms [15]; and indirect branch instructions such as `call/jmp reg32` that make it difficult or impossible to identify the target location [10, 14]. Due to these issues and others, binary analysis/rewrite tools for the x86 platform have strict restrictions on their applicable targets [9, 10, 14, 15]. As such, we have deferred consideration of these techniques in our framework for the time being.

Other omissions from our present study are system call arguments (a topic of ongoing work) and other paradigms that have been proposed for detecting when a process has been commandeered via the insertion of foreign code into the process address space (e.g., program shepherding [8]).

3 Empirical study of gray-box program tracking

The parameters Σ and n are central to the effectiveness of an anomaly detection system. Together these parameters determine the states of the automaton, and thus the history information on which the automaton "decides" that a new input $\sigma \in \Sigma$ is anomalous. Intuitively, increasing the information in each element of Σ or n increases the number of states of the automaton, and thus the granularity and accuracy of anomaly detection. In this paper we view this greater sensitivity as a benefit, even though it comes with the risk of detecting more anomalies that are not, in fact, intrusions. However, since we restrict our attention to techniques that ensure that any transition (triggered by sys-

n	Σ					
	S	**P**	**R**	$\mathbf{S}^+$	$\mathbf{P}^+$	$\mathbf{R}^+$
0				[23, 24] $\surd$	$\surd$	$\surd$
1	[4] $\surd$	[16] $\surd$	[3] $\surd$	$\surd$	$\surd$	$\surd$
≥ 2	[4] $\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$

Table 1: Scope of this paper ($\surd$) and prior work

tem call sequences) in the training data will never result in a transition to $q_\perp$, we simply assume that our detectors are adequately trained and consider this risk no further. As such, the primary costs we consider for increasing each of these parameters are the additional overhead for collecting information and the size of the transition relation δ.

Our goal in this section is to provide a systematic analysis of the costs and benefits of enhancing these parameters. Specifically, we study the following question: For given costs, what combination of Σ and n is most beneficial for anomaly detection? We reiterate that as shown in Table 1, this study introduces several new possibilities for anomaly detection that, to our knowledge, have not yet been studied.

3.1 Mimicry attacks

To understand the benefits of growing Σ or n, it is necessary to first understand the principles behind mimicry attacks [18, 21]. An attack that injects code into the address space of a running process, and then causes the process to jump to the injected code, results in a sequence of system calls issued by the injected code. In a mimicry attack, the injected code is crafted so that the "attack" system calls are embedded within a longer sequence that is consistent with the program that should be running in the process. In our model of Section 2, this simply means that the attack issues system calls that avoid sending the automaton to state $q_\perp$.

There are many challenges to achieving mimicry attacks. First, it is necessary for the injected code to forge all information that is inspected by the anomaly detector. This seems particularly difficult when the anomaly detector inspects the program counter and all return addresses in the process call stack, since the mechanics of program execution would seem to force even the injected code to conform to the program counter and stack it forges in order to make a system call (which must be the same as those in the correct process to avoid detection). Nevertheless, we demonstrate in Section 4 that mimicry remains possible. While we are not concerned with the mechanics of doing so for the present section, we do wish to analyze the impact of monitoring program counter and return address information on these attacks. Specifically, in order to forge this information, the injected attack code must incorporate the address information to forge (possibly compressed), and so this necessarily increases the size of the attack code. As such, a goal of our analysis is to quantify the increase in size of the attack code that results from the burden of carrying this extra information. We comment that this size increase can impose upon the viability of the attack, since the area in which the injected code is written is typically bounded and relatively small.

A second challenge to achieving a mimicry attack is that a step of the attack may drive the automaton to a state that requires a long sequence of intervening system calls to reach the next system call in the attack, or that even makes reaching the next system call (undetected) impossible. In general, enhancing Σ or growing n increases this challenge for the attacker, as it increases the granularity of the state space. This must be weighed against the increased size of the automaton, however, as well as the additional run-time costs to extract the information dictated by Σ. A second aspect of our analysis in this section is to explore these tradeoffs, particularly with an eye toward $|\delta|$ as the measure of automaton size.

3.2 Analysis

In order to analyze the costs and benefits of enhancing the axes of the state space, we set up a testbed anomaly detection system. The system is implemented as a kernel patch on a Red Hat Linux platform, with configuration options for different values of Σ and n. We implement the variable-length pattern approach as described in [13, 24] for each $\Sigma \in \{\mathbf{S}^+, \mathbf{P}^+, \mathbf{R}^+\}$. We have chosen four common FTP and HTTP server programs, `wu-ftpd`, `proftpd`, Apache `httpd`, and Apache `httpd` with a `chroot` patch, for evaluation purposes. Automata for these four programs (and different configurations of the axes) are obtained by training the anomaly detection system with between four and nine million of system calls generated from test runs. After obtaining the automata, we perform analysis to evaluate the costs and benefits of different configurations of Σ and n. Figures 1 and 2 show the results when $\Sigma \in \{\mathbf{S}, \mathbf{P}, \mathbf{R}\}$ and $\Sigma \in \{\mathbf{S}^+, \mathbf{P}^+, \mathbf{R}^+\}$, respectively. That is, Figures 1 and 2 correspond to the two possible instantiations of axis 2 in Section 2.

3.2.1 Resilience against mimicry attacks

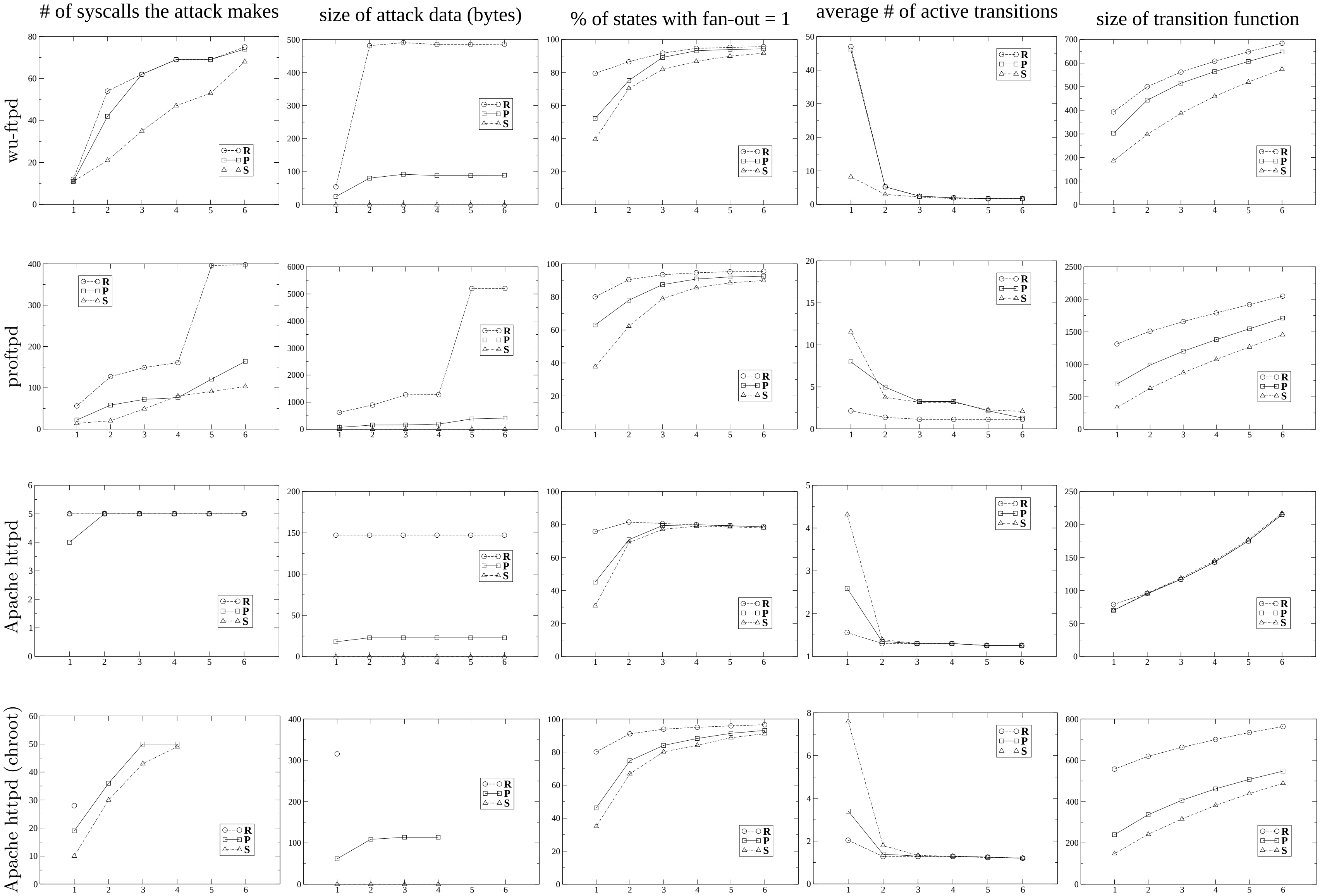
The first three columns of Figures 1 and 2 are about resilience against mimicry attacks. The attack we test is the addition of a backdoor root account into the password file. This common attack needs to perform a series of six system calls (`chroot`, `chdir`, `chroot`, `open`, `write`, `close`), which is similar to the attack sequence discussed in [21]. However, in the case of Apache `httpd` only three system calls are needed (`open`, `write`, `close`). We choose to analyze this attack sequence because it is one of the most commonly used system call sequences in an attack. Many attacks need to make system calls that constitute a superset of this sequence.

We perform an exhaustive search to find the shortest sequence containing the above series of system calls, not necessarily contiguously, that avoids detection.[5] The exhaustive search reveals the best an attacker can do to evade detection when making the attack system calls. Graphs on the first column show the minimum number of system calls a mimicry attack must make in order to evade detection. (Missing data points on the graphs indicate that the mimicry attack is not possible.) For example in the case of Apache `httpd` with `chroot` patch, the mimicry attack makes 28 system calls when $\Sigma = \mathbf{R}$ and $n = 1$, while it becomes impossible for $n \geq 2$ with the same setting of Σ. It is clear from the graphs that growing Σ or n makes mimicry attacks more difficult.

It might not be obvious why the mimicry attack becomes impossible when $\Sigma = \mathbf{R}$ while it is possible for $\Sigma = \mathbf{P}$ with the same setting of n. (For example, the graph of Apache `httpd` (`chroot`) in the first column of Figure 1 shows that the mimicry attack is impossible when $\Sigma = \mathbf{R}$ and $n \geq 2$.) Here we explain with a simple example. In Figure 3, a solid rectangle represents a state in the automaton, and r, p and s represent a set of return addresses, a program counter and a system call number respectively. If the anomaly detector does not check return addresses, the two states (r_1, p, s) and (r_2, p, s) will collapse into one and the impossible path denoted by the dashed line will be accepted, which makes a mimicry attack possible. Thus, checking return addresses makes the automaton model more accurate.

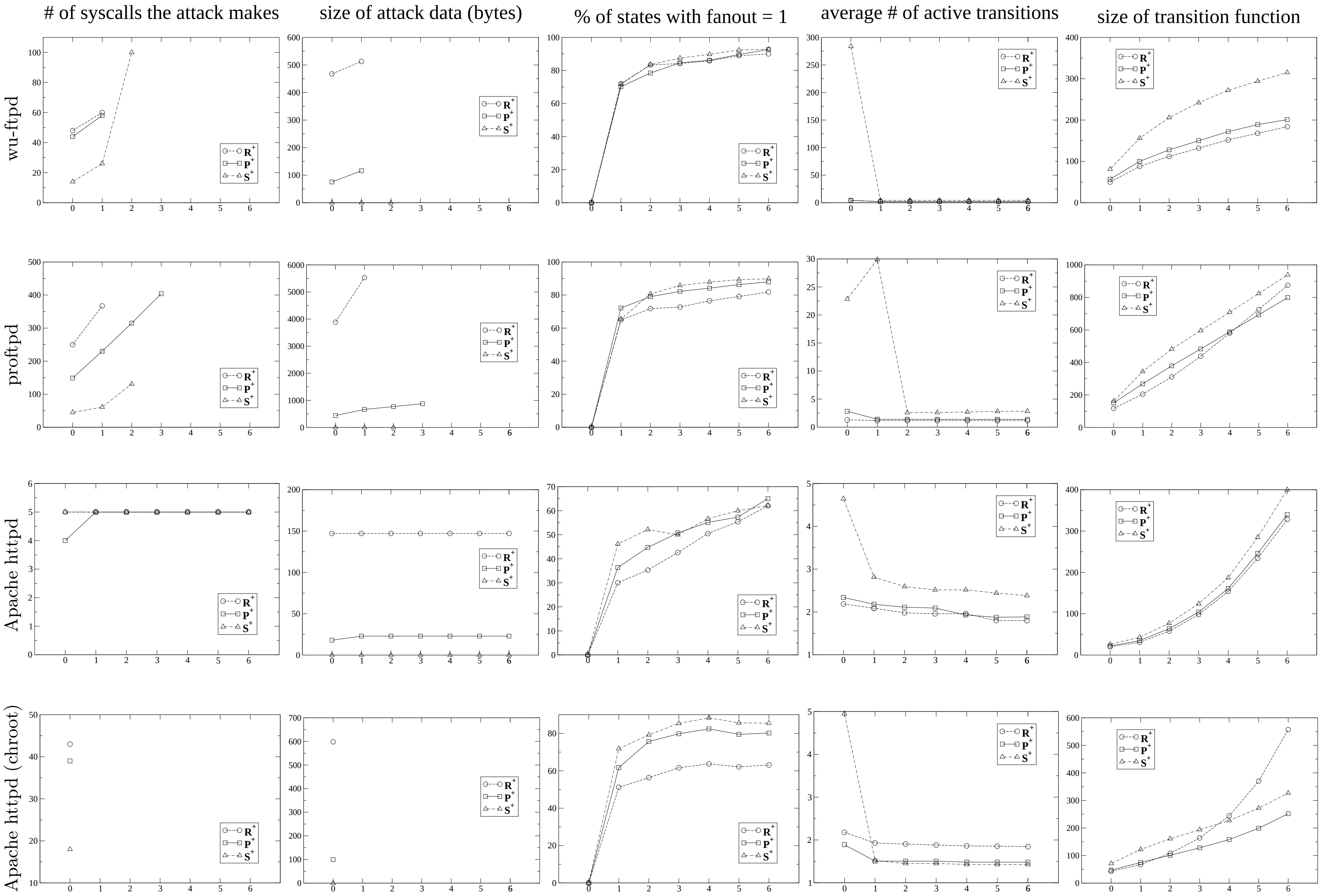
Although the minimum number of system calls an attack makes is a good measure of the difficulty of a mimicry attack, in many cases attackers are free to make any number of system calls, as long as they do not set off any alarms in the anomaly detector. However, in the case where $\Sigma \in \{\mathbf{P}, \mathbf{R}, \mathbf{P}^+, \mathbf{R}^+\}$, the attack has to forge all information that is inspected by the anomaly detection system (program counters and return addresses). We thus provide a second measure on the difficulty of the mimicry attack, namely the size of the attack data, which is shown by the graphs on the second column of Figures 1 and 2. In this measure we only take into account the attack data, which is the forged program counter and the return addresses (and nothing in the case of $\mathbf{S}$ and $\mathbf{S}^+$), with the assumption of perfect compression. Again the graphs show that growing Σ or n makes mimicry attacks consume significantly more space. Note that the size increase in attack data could make mimicry attacks less efficient (due to the need to send more data), easier to detect, or even make some mimicry attacks impossible due to limited space in the program buffer where the attack code is inserted. For example, the size of the attack data becomes a few kilobytes on the `proftpd` program in some configurations.

The analysis so far has been focused on one mimicry attack. In an effort to quantify the difficulty of mimicry attacks in general, we define a property of an automaton state, called its *fanout*, as follows: $\text{fanout}(q) = |\delta(q)|$, where $\delta(q) := \{(q, \sigma, q') | (q, \sigma, q') \in \delta\}$. $\text{fanout}(q)$ measures the number of possible states that can follow an active state q. If an attack compromises the program and, in the course of performing its attack, activates q, then only $\text{fanout}(q)$ states can follow from q. As such, $\text{fanout}(q)$ is a coarse measure of the extent to which a mimicry attack is constrained upon activating q. Graphs in the third column of Figures 1 and 2 show the percentage of states with $\text{fanout}(q) = 1$ in



(x-axis in all graphs above represents the value of n, legends show the configuration of Σ.)

Figure 1: Evaluation results on $\Sigma = \mathbf{S}$, $\mathbf{P}$ and $\mathbf{R}$ with varying window size n



(x-axis in all graphs above represents the value of n, legends show the configuration of Σ.)

Figure 2: Evaluation results on $\Sigma = \mathbf{S}^+$, $\mathbf{P}^+$ and $\mathbf{R}^+$ with varying window size n

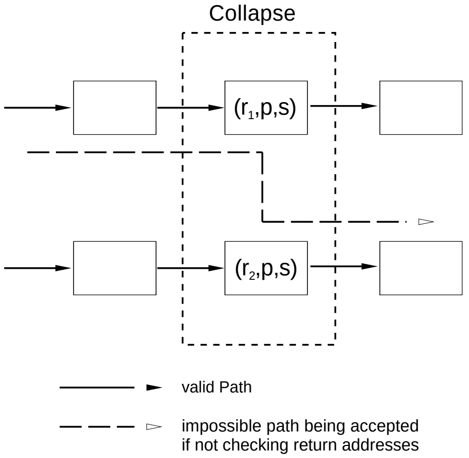

Collapse
(r_1,p,s)
(r_2,p,s)
valid Path
impossible path being accepted
if not checking return addresses

Figure 3: Two states collapse if return addresses are not checked

each automata. As seen from the graphs, the percentage of states with fanout$(q) = 1$ increases as n increases, especially when n is small.

We note that average branching factor as introduced in [20] is a conceptually similar measure. Here we prefer to use *fanout* because *fanout* measures the property of an automaton, whereas average branching factor is a property of executions of the program, as well. Another difference is that *fanout* considers all possible transitions regardless of whether the system call that triggers it is "harmful" as determined in [20] or not. Thus for any particular automaton, *fanout* should have a much higher value than average branching factor, which is used in [5, 6, 20].

3.2.2 Overhead

The previous three measures give evidence that growing Σ or n makes mimicry attacks more difficult. However, doing so also increases the cost of the anomaly detector. We would thus like to measure the performance overhead in order to find the best configuration of Σ and n.

The first measure we evaluate is the cost of extracting program counters and return addresses. We run two sets of tests, one with and one without the Linux kernel configured to extract return addresses from the process when a system call is made, and measure the time it takes to do a Linux kernel compilation. Results (Table 2) show that the performance hit is especially noticeable in the system time, which measures the time spent in the kernel. However, this translates to less than 6% increase in the overall execution time. Therefore, utilizing $\Sigma \in \{\mathbf{P}, \mathbf{R}, \mathbf{P}^+, \mathbf{R}^+\}$ introduces only moderate overhead.

We next consider the amount of processing the anomaly detector has to do when a system call is made. At any point in time, the anomaly detector must track the active states $q \in \mathcal{Q}$, as well as the transitions that the next input symbol from Σ may trigger ("active transitions"). When a system call is made, active transitions are examined to determine the next active states and next active transitions.[6] We simulate executions of the FTP and HTTP server programs and measure the number of active transitions whenever a system call is made. Finally we calculate the average of these figures and present them in the fourth column of Figures 1 and 2. As shown in these graphs, growing Σ or n reduces the number of active transitions and thus the processing time of the anomaly detection system. Another observation is that when $n \geq 3$, increasing n seems to have less effect and the active number of transitions becomes very close to one.

Memory usage and storage overhead is another important measure of performance. As a coarse measure of the storage overhead, here we calculate $|\delta|$ for each of the automata; the results are pictured in the last column of Figures 1 and 2. Intuitively, growing Σ or n should increase the size of δ, due to the increase in granularity and accuracy of the automaton. This is confirmed by graphs in Figure 1. However, graphs in the last column of Figure 2 suggest opposite results, as the size of transition function of $\Sigma = \mathbf{R}^+$ is less than those of $\Sigma = \mathbf{P}^+$ and $\Sigma = \mathbf{S}^+$ for some values of n. A closer look at the automata reveals that the average length of $\sigma \in \Sigma$ (number of system calls in an atomic unit) is larger in the case $\Sigma = \mathbf{R}^+$ than it is when $\Sigma \in \{\mathbf{S}^+, \mathbf{P}^+\}$, leading to a reduced number of states and a smaller transition relation for some values of n. This is true for all four FTP and HTTP programs in our implementation of the pattern extraction algorithm. However, whether this holds for other pattern extraction algorithms remains future work.

3.3 Discussion and recommendations

Looking at the first axis (runtime information captured by the anomaly detector), we observe that

		no checking (seconds)	checking (seconds)
average of 3 tests	overall	80.205	84.934
	user	66.397	66.917
	system	13.103	16.633
average overhead	overall		5.896 %
	user		0.783 %
	system		26.940 %

Table 2: Performance overhead for checking return addresses

checking return addresses ($\Sigma \in \{\mathbf{R}, \mathbf{R}^+\}$) greatly increases the difficulty of mimicry attacks. Although these addresses could possibly be forged by attackers (see Section 4), it requires not only detailed understanding of the vulnerable program and its automaton, but also careful crafting of the attack code and sufficient buffer size for it. Since the performance overhead for checking return addresses is moderate (Table 2), an anomaly detection system should always check return addresses.

As for the second axis, the evidence suggests that forming atomic units from variable-length subsequences makes mimicry attacks difficult even with a small value of n. This is an interesting result, as a small value of n indicates smaller memory usage and storage overhead (last column of Figure 2). Although $\Sigma \in \{\mathbf{S}^+, \mathbf{P}^+, \mathbf{R}^+\}$ introduces nondeterminism into the automaton (supposing that the technique of [13, 24] is used), with $n \geq 2$ there are fewer than two active transitions on average, and thus the system processing time should be sufficiently small.

The third axis (value of n) shows some tradeoff between accuracy and performance. Since increasing n has little effect on improving accuracy when $\Sigma = \mathbf{R}^+$ and $n \geq 2$ (refer to the first 4 columns in Figure 2), we consider the setting of $\Sigma = \mathbf{R}^+$ and $n = 2$ as a general recommendation, which makes mimicry attacks difficult with reasonably low costs in performance. (Some complicated programs might require n to take a slightly bigger value, with an increase in performance overhead.)

However, choosing $\Sigma \in \{\mathbf{S}^+, \mathbf{P}^+, \mathbf{R}^+\}$ requires an extra step in constructing the automaton, which is to extract the variable-length patterns. Different parameter settings in the pattern extraction algorithm could yield very different results. It remains future work to analyze the best pattern extraction algorithm and its parameter settings. Nevertheless, our relatively simple implementation of the pattern extraction algorithm produces very promising results for monitoring accuracy and performance.

4 Program counter and return address forgery

A presumption for the analysis of Section 3 was that an attacker is able to forge the program counter and return addresses of the process execution stack. In a gray-box monitoring approach, these values are extracted by the monitor automatically per system call, by directly examining the relevant portions of the process address space. As such, these values constitute state that controls the subsequent execution of the process upon return of the system call from the kernel, due to the mechanics of process execution. It is therefore not obvious that an attack could effectively forge these values: For example, the first system call of the attack would seemingly return control to the program that the process should be running. Indeed, prior work that proposed monitoring return addresses [3] largely discarded the possibility that these values could be undetectably forged.

In this section we describe how these values can, in fact, be undetectably forged. We describe this attack for the Linux execution environment, though our approach can be generalized to other environments, as well. The principle behind our attack is to modify the stack frame, so that the detector does not observe an anomaly, even for system calls made by the attack code. (Please refer to Appendix A for a brief review on the structure of a stack frame.)

We demonstrate our attack using a very simple victim program; see Figure 4. We emphasize that we have implemented successful attacks for the program in Figure 4 against (our own implementations of) the anomaly detection techniques of [3, 16], as well as against an independent implementation of return address monitoring by the authors of that technique [3]. The victim program takes a command line argument and passes it to `f1()`. `f1()` calls another function `f2()` twice, which calls a library function `lib()`. The function `lib()` in the victim program makes a system call, with `17` as the system call number. Function `f2()` is called twice just to make the victim program have multiple system calls. The victim program is designed in this way to demonstrate how most software programs make system calls. Note that `f1()` has a local buffer that can be overflowed.

```
void lib() { syscall(17); }

void f2() { lib(); }

void f1(char* str) { char buffer[512];
                     f2(); f2();
                     strcpy(buffer,str); }

int main(int argc, char *argv[]) {
   f1(argv[1]); }
```

Figure 4: C source code of victim program

4.1 Forging the program counter

Upon receiving a system call from the monitored process, the program counter indicates the address of the instruction initiating the system call. Since most system call invocations are made from within a library function in `libc` (`lib()` in our sample victim program in Figure 4), the value of the program counter is often not useful, particularly for dynamically linked libraries. Therefore, in the work that introduced monitoring the program counter, Sekar et al. [16] instead trace back each system call to the most recent function invocation from the statically linked code section, and use this location as the program counter. By doing this, the program counter value will be an address in the program that results in the system call, rather than an address in the library. We take a similar approach in our work. Before the program is run, the anomaly detection system examines the section header table of the binary executable to find out address range of the code (text) section.[7] At runtime, it determines the program counter by tracing the return addresses from the innermost stack frame until a return address falls within that address range.

In order to evade detection by such a monitor, an attack should ensure that:

1. The address of the attack code does not appear as a return address when the anomaly detector is tracing the program counter.

2. The program counter found by the anomaly detection system is a valid address for the system call made.

Because of the first requirement, our attack code cannot `call` a library function to make system calls. If the attack code uses a `call` instruction, the address of the attack code will be pushed onto the stack and the anomaly detection system will observe the anomaly. So, instead of using a `call` instruction, our attack code uses a `ret` instruction. (A `jump` instruction could serve the same purpose.) The difference between a `call` and a `ret` instruction is that the `call` instruction pushes the return address onto the stack and then jumps to the target location, whereas a `ret` instruction pops the return address and then jumps to that location. If we can make sure that the return address is the address of the instruction in the library function that makes the corresponding system call, we could use the `ret` instruction in place of a `call` instruction. Figure 5a shows the stack layout right before the `ret` instruction is executed. By forging this stack frame, the address of an instruction in `lib()` will be used as the return address when `ret` is executed.

In order to satisfy the second requirement, we must forge another address on the stack, which the monitor will determine to be the location where `lib()` is called. Our attack code simply inserts a valid address (i.e., one that the monitor will accept for this system call) at the appropriate location as the forged program counter. Figure 5b shows the stack layout after the first `ret` is executed, as seen by the anomaly detection system.

As described previously, the above suffices to achieve only one system call of the attack: after it has been completed, control will return to the code indicated by the forged program counter. However, most attacks need to make at least a few system calls. Thus we have a third requirement.

3. Execution will return to attack code after an attack system call finishes.

	...
	addr of instruction in `lib()`
`ebp` →	old `ebp`
`esp` →	...

(a) Before `ret` is executed

	...
	argument of `lib()`
	forged program counter
`ebp` →	old `ebp`
`esp` →	...
	addr of instruction in `lib()` (popped)
	old `ebp` (popped)

(b) After `ret` is executed

	...
	argument of `f1()`
	addr of last instruction in `main()`
	old `ebp`
	`buffer`
	...
	addr of `strcpy()` in `f1()`
`ebp` →	old ebp
`esp` →	...
	argument of `lib()` (popped)
	forged program counter (popped)
	old `ebp` (popped)
	... (popped)
	addr of instruction in `lib()` (popped)
	old `ebp` (popped)

(c) After `lib()` returns

Figure 5: Stack layouts in program counter forgery (stack grows downwards)

The idea to achieve this is to modify a return address remaining on the stack after the system call finishes. However, a challenge is that the instruction that does this modification has to be an instruction in the original program's code, because at that time execution has not returned to the attack code yet. Generally speaking, any instruction that performs assignment by pointer dereferencing could be used. For example if `a` is defined as `long*`, and `b` is defined as `long`, the instruction `*a = b;` could be used for our purpose. We just need to modify the stack, including the `ebp` value, so that `a` is the address of the return address that we want to modify, and `b` is the value of an address in the attack code. Such assignment instructions are common in C programs.

In our victim program (Figure 4) there is no instruction that performs simple assignment by pointer dereferencing like `*a = b;`. We implement our attack in a different way. In the victim program, the call to `strcpy()` is used to overflow the buffer and therefore overwrite the return address. We could execute this instruction again when the first system call made by the attack code finishes. This overflows the buffer and overwrites the return address again. Execution will return to the attack code when `f1()` returns.

Figure 5c shows the stack layout our attack has to forge in order to satisfy all three requirements. Execution will return to `strcpy()` in `f1()` and by doing that, the return address of `f1()` will be overwritten again. This ensures that execution will go back to the attack code after a system call is made. Since execution always starts at the same location in the attack code, we need to keep some state information. This could be realized by a counter. Each time the attack code is entered the counter is checked and incremented, so that the attack code knows how many system calls it has already made.

4.2 Forging return addresses

We have also successfully extended our attack to anomaly detection systems that monitor the entire set of return addresses on the stack. The attack is confirmed to be successful against an implementation of anomaly detection approach proposed by Feng et al. [3].

To achieve this, we need to modify our attack only slightly to forge the entire set of return addresses on the process execution stack. In the attack described in Section 4.1, we forged one return address so that the monitor will see a valid program counter value. Here, the attack is simply required to forge more stack frames, including that for `main()`. The forgery is simpler in this case, however, as the stack frames contain only the return address and the old `ebp` value, without any arguments or local variables. These stack frames are only checked by the anomaly detection system, and they are not used in program execution at all.

5 Using randomization to defend against forgery attacks

In this section we propose a (white-box) randomization technique to defend against the forgery attack presented in Section 4. The attack of Section 4 requires the attacker to have an in-depth understanding of the internal details of the victim program, as well as the automaton representing the normal behavior of the victim program; e.g., the attacker must know the value of the program counter and return addresses to forge. Thus, randomization techniques could be used to render this type of attack more difficult.

Although there have been previous works on address obfuscation, e.g., [1], our goal here is to hide program counter and return address values and prevent attackers from forging them, which is different from previous works. Kc et al. [7] introduce the idea of randomizing the instruction set to stop code-injection attacks. However, our randomization technique does not require special processor support as required in [7].

An initial attempt is to randomize a base address. Two types of base addresses could be randomized: the starting address of dynamically linked libraries and the starting address of the code segment in the executable. The former can be implemented by inserting a dummy shared library of random size. The latter can be implemented by simple modifications to the linker. Changes to these base addresses are easy to implement. However, this randomization relies on only a single secret.

A more sophisticated technique is to reorder functions in the shared library and/or the executable. This can be combined with the first approach to introduce a different random offset for each function, although implementation becomes a bit more complicated. Both above techniques rely on the availability of the object code.

Although white-box approaches could be problematic on x86 platform as discussed in Section 2, reordering functions in the dynamically linked library and/or the executable is not difficult for the following reasons. First, we do not need to make any changes within a function block. Most other white-box techniques (e.g., [5, 6, 10]) need to analyze individual instructions in function blocks and insert additional instructions. Second, since the section header table is always available for relocatable files (not true for executables) and the dynamic symbol table is always available for shared libraries, binary analysis becomes much easier.

We note, however, that even this defense is not foolproof: if the attacker is able to view the memory image of the running process, the randomized addresses could be observed. As such, the attacker's code running in the address space of the process could scan the address space to discern the randomized addresses and then adjust the return addresses it forges on the call stack accordingly. However, this substantially complicates the attack, and possibly increases the attack code size.

6 Conclusions and future work

In this paper we perform the first systematic study on a wide spectrum of anomaly detection techniques using system calls. We show that previous proposed solutions could be organized into a space of three axes, and that such an organization reveals new possibilities for system-call-based program tracking. We demonstrate through systematic study and empirical evaluation the benefits and costs of enhancing each of the three axes and show that some of the new approaches we explore offer better properties than previous approaches. Moreover, we demonstrate novel mimicry attacks on a recent proposal using return addresses for system-call-based program tracking. Finally we describe how a simple white-box randomization technique can make such mimicry attacks more difficult.

We have analyzed the program counter and return addresses as the runtime information acquired by the anomaly detector. Other runtime information we have not considered is the system call arguments. It remains future work to include system call arguments in our systematic analysis. The pattern extraction algorithm used to group related system calls together as an atomic unit is another area that requires further attention.

Acknowledgements

This work was partially supported by the U.S. Defense Advanced Research Projects Agency and the U.S. National Science Foundation.

Notes

[1] Prior work [3] states only that "... the intruder could possibly craft an overflow string that makes the call stack look not corrupted while it really is, and thus evade detection. Using our method, the same attack would probably still generate a virtual path anomaly because the call stack is altered." Our attack demonstrates that this trust in detection is misplaced.

[2] m ranges from 1 to n because the number of atomic units the anomaly detector remembers is less than n in the first n states of program execution.

[3] In [17], n is recommended to be 6, which corresponds to $n = 5$ in our parlance.

[4] Prasad and Chiueh claim that this renders the problem of distinguishing code from data undecidable [10].

[5] Our exhaustive search guarantees that the resulting mimicry attack involves the minimum number of system calls made in the case of `wu-ftpd`, Apache `httpd` and Apache `httpd` with `chroot` patch. However due to the complexity of the `proftpd` automaton, we could only guarantee minimum number of intervening system calls between any two attack system calls.

[6] If the automaton is, in fact, deterministic, then optimizations are possible. In this analysis we do not explicitly consider these optimizations, though the reader should view the fourth column of Figure 1 as potentially pessimistic.

[7] Strictly speaking, this constitutes white-box processing, though qualitatively this is distant from and far simpler than the in-depth static analysis performed by previous white-box approaches. Were we to insist on sticking literally to gray-box techniques, however, we could extract the same information at run time using less convenient methods.

References

[1] S. Bhatkar, D. DuVarney and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceeding of the 12th USENIX Security Symposium*, pages 105–120, August 2003.

[2] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.

[3] 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*, pages 62–75, May 2003.

[4] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, May 1996.

[5] J. Giffin, S. Jha and B. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, August 2002.

[6] J. Giffin, S. Jha and B. Miller. Efficient context-sensitive intrusion detection. In *Proceeding of Symposium on Network and Distributed System Security*, Febuary 2004.

[7] G. Kc, A. Keromytis and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceeding of the 10th ACM Conference on Computer and Communication Security*, pages 272–280, October 2003.

[8] V. Kiriansky, D. Bruening and S. Amarasinghe. Secure execution via program shepherding. In *Proceeding of the 11th USENIX Security Symposium*, August 2002.

[9] X. Lu. A Linux executable editing library. Master's Thesis, Computer and Information Science, National Unviersity of Singpaore. 1999.

[10] M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *USENIX Annual Technical Conference, General Track*, June 2003.

[11] N. Provos. Improving host security with system call policies. In *Proceeding of the 12th USENIX Security Symposium*, August 2003.

[12] N. Provos, M. Friedl and P. Honeyman. Preventing privilege escalation. In *Proceeding of the 12th USENIX Security Symposium*, August 2003.

[13] I. Rigoutsos and A. Floratos. Combinatorial pattern discovery in biological sequences: the TEIRESIAS algorithm. In *Proceedings of the 1998 Bioinformatics*, vol. 14 no. 1, pages 55–67, 1998.

[14] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad and B. Chen. Instrumentation and optimization of Win32/Intel executables using etch. In *Proceeding of the USENIX Windows NT workshop*, August 1997.

[15] B. Schwarz, S. Debray and G. Andrews. Disassembly of executable code revisited. In *Proceeding of Working Conference on Reverse Engineering*, pages 45–54, Oct 2002.

[16] R. Sekar, M. Bendre, D. Dhurjati and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 144–155, May 2001.

[17] K. Tan and R. Maxion. "Why 6?"—Defining the operational limits of stide, an anomaly-based intrusion detector. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 188-201, May 2002.

[18] K. Tan, J. McHugh and K. Killourhy. Hiding intrusions: from the abnormal to the normal and beyond. In *Proceedings of Information Hiding: 5th International Workshop*, pages 1–17, January 2003.

[19] D. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, Department of Computer Science, University of California at Berkeley, August 1999.

[20] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 156–168, May 2001.

[21] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, November 2002.

[22] R. Wahbe, S. Lucco, T. E. Anderson and S. L. Graham. Efficient software-based fault isolation. In *Proceeding of the Symposium on Operating System Principles*, 1993.

[23] A. Wespi, M. Dacier and H. Debar. An intrusion-detection system based on the Teiresias pattern-discovery algorithm. In *Proceedings of the 1999 European Institute for Computer Anti-Virus Research Conference*, 1999.

[24] 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*, pages 110–129, October 2000.

A Review of stack frame format

The call stack of the system we are using in this paper is divided up into contiguous pieces called stack frames. Each frame is the data associated with a call to one function. The frame contains the arguments given to the function, the function's local variables, etc. When the program is started, the stack has only one frame. Each time a function is called, a new frame is made. Each time a function returns, the frame for that function invocation is eliminated. If a function is recursive, there can be many frames for the same function. The frame for the function in which execution is actually occurring is called the *innermost* frame.

The layout of a stack frame is shown in Figure 6. `ebp` always stores the address of the old `ebp` value of the innermost frame. `esp` points to the current bottom of the stack. When program calls a function, a new stack frame is created by pushing the arguments to the called function onto the stack. The return address and old `ebp` value are then pushed. Execution will switch to the called function and the `ebp` and `esp` value will be updated. After that, space for local variables are reserved by subtracting the `esp` value. When a function returns, `ebp` is used to locate the old `ebp` value and return address. The old `ebp` value will be restored and execution returns to the caller function.

	function arguments
	return address
`ebp` →	old `ebp`
`esp` →	local variables

Figure 6: Stack frame layout (stack grows downwards)

B Source code for attack in Section 4

```
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             545
#define NOP                            0x90

char attackcode[] =
"\x5d"                          /* pop   %ebp                         */
"\x68\x81\xf9\xff\xbf"          /* push  bffff987 (arg to f1)         */
"\x68\x42\x86\x04\x08"          /* push  8048642 (forge ret addr)     */
"\x83\xec\x7f"                  /* sub   $0x7f, %esp                  */
"\x83\xec\x7f"
"\x83\xec\x7f"
"\x83\xec\x7f"
"\x83\xec\x7f"
"\x83\xec\x7f"
"\x68\xe5\x85\x04\x08"          /* push  80485e5 (after f2 in f1)     */
"\x68\xd8\xf7\xff\xbf"          /* push  bffff7d8 (correct ebp of f1) */
"\x89\xe5"                      /* mov   %esp,%ebp                    */
"\x68\x47\x85\x04\x08"          /* push  8048547 (end of f2)          */
"\x55"                          /* push  %ebp                         */
"\x89\xe5"                      /* mov   %esp,%ebp                    */
"\x68\xd3\x84\x04\x08"          /* push  80484d3 (start of f3/lib)    */
"\x55"                          /* push  %ebp                         */
"\x89\xe5"                      /* mov   %esp,%ebp                    */
"\xc9"                          /* leave                              */
"\xc3";                         /* ret                                */


int main(int argc, char *argv[]) {
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (!(buff = malloc(bsize))) {
    printf("Can’t allocate memory.\n");
    exit(0);
  }

  addr = 0xbffff5d0;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;

  /* return address */
  for (i = 0; i < bsize; i+=4)
    *(addr_ptr++) = addr;

  /* no-op */
  for (i = 0; i < bsize/2; i++)
    buff[i] = NOP;

  /* attack code */
  ptr = buff + ((bsize/2) - (strlen(attackcode)/2));
  for (i = 0; i < strlen(attackcode); i++)
    *(ptr++) = attackcode[i];

  /* restore ebp */
  ptr = buff + bsize - 9;
  addr_ptr = (long *)ptr;
  *(addr_ptr) = 0xbffff7f8;

  /* end of string */
  buff[bsize - 1] = ’\0’;

  execl("./victim", "victim", buff, 0);
}
```