International Conference on Automonic Computing

Anatomy of a Real-time Intrusion Prevention System

Ricardo Koller, Raju Rangaswami, Joseph Marrero, Igor Hernandez, Geoffrey Smith,
Mandy Barsilai, Silviu Necula, S. Masoud Sadjadi, Tao Li, and Krista Merrill
School of Computing and Information Sciences, Florida International University

***Abstract*—Host intrusion prevention systems for both servers and end-hosts must address the dual challenges of *accuracy* and *performance*. Researchers have mostly focused on addressing the former challenge, suggesting solutions based either on exploit-based penetration detection or anomaly-based misbehavior detection, but yet stopping short of comprehensive solutions that leverage merits of both approaches. The second challenge, however, is rarely addressed; doing so comprehensively is important since these systems can introduce substantial overhead and cause system slowdown, more so when the system load is high.**

We present Rootsense, a holistic and real-time intrusion prevention system that combines the merits of misbehavior-based and anomaly-based detection. Four principles govern the design and implementation of Rootsense. First, Rootsense audits events within different subsystems of the host operating system and correlates them to comprehensively capture the global system state. Second, Rootsense restricts the detection domain to root compromises *only*; doing so reduces run-time overhead and increases detection accuracy (root behavior is more easily modeled than user behavior). Third, Rootsense adopts a dual approach to intrusion detection – a *root penetration detector* detects activities that exploit system vulnerabilities to penetrate the security perimeter, and a *root misbehavior detector* tracks misbehavior by root processes. Fourth, Rootsense is designed to be configurable for overhead management allowing the system administrator to tune the overhead characteristics of the intrusion prevention system that affect foreground task performance. A Linux implementation of Rootsense is analyzed for both accuracy and performance, using several real-world exploits and a range of end-host and server benchmarks.

I. Introduction

Intrusion detection and prevention systems must address two challenges comprehensively: *accuracy* and *performance*. Accuracy is concerned with both *false positives* as well as *false negatives* of the intrusion detection mechanism, while performance is measured by impact to the foreground task *response time distribution*. Each of these metrics affect the end-user experience and are critical factors determining whether the solution is practical for use in production systems.

There has been substantial research on addressing accuracy and current solutions can be classified at the high-level into *exploit-based penetration detection* [1], [2], [3] or *anomaly-based misbehavior detection* [4], [5], [6]. It is well-accepted that while the former approach can be susceptible to false negatives, the latter must deal with (often large number of) false positives. However, little research exists on how we can combine these complementary techniques to effectively increase accuracy. Solutions focusing on accuracy also often target a specific subsystem. Systems such as StackGuard [7], Tripwire [8], Snort [9], and Bro [10] rely on monitoring a single subsystem (e.g., *memory* or *filesystem*) to derive inferences. However, as we shall exemplify later on, many intrusions as well as post-intrusion misbehavior involve a sequence of operations affecting *multiple* subsystems. Finally, most solutions addressing the former challenge attempt to detect *all* intrusions in the system, while some focus on detecting *root domain* intrusions only [5], [11], [12]. This choice can impact the efficiency, accuracy, and usability of the system.

The latter challenge, namely *performance*, has received relatively less attention. If not addressed comprehensively, performance can be the roadblock to large-scale adoption of real-time intrusion prevention solutions. Specifically, the overhead associated with monitoring (e.g., data collection), analysis (e.g., signature-matching), and response, in terms of their impact to foreground tasks are not well understood. Here, the *response time distribution* is a key metric that captures both the *average response time* and *jitter* experienced by foreground tasks. What is required is a control mechanism that will allow the system administrator to monitor and tune the response time distribution (to whatever extent possible) to reflect overall system performance goals.

In this paper, we present Rootsense, a real-time intrusion prevention system, which monitors *multiple subsystems* to get comprehensive information about the global state of the host, focuses on the restricted but critical domain of *root intrusions*, and employs a *dual approach* for detecting both vulnerability-specific exploits as well as anomalous processes. Without sacrificing its accuracy, Rootsense also provides overhead tunability that the system administrator can use to control the execution behavior of the intrusion prevention system. Consequently, the administrator can control impact on foreground performance, specifically allowing a trade-off between foreground task average response time and jitter.

The design of Rootsense addresses several challenges, including minimizing overhead and exporting control over the system overhead characteristics, addressing the scope and detail of the information collected, correlating the collected information within and across subsystems, generating activity signatures of interest at the right level of abstraction, using such signatures for accurate and efficient intrusion detection, and creating a timely and configurable response mechanism.

The design, implementation, and a detailed performance analysis of Rootsense are the subject of the remainder of the paper. Section II discusses the design principles adopted in Rootsense and gives a high-level description of its architecture. Section III defines the fundamental concepts of *events*, *activities*, and *signatures*. Section IV describes the intrusion

978-0-7695-3175-5 2008
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/ICAC.2008.24

IEEE computer society

detection algorithms used by Rootsense. Section V presents an experimental evaluation of Rootsense. Section VI discusses related work and we conclude in Section VII.

II. Design Principles and System Architecture

A. Design Principles

Rootsense's design is directed by four key principles:

A. Holistic monitoring. We gain a more comprehensive view of the global system state by monitoring four subsystems: Process, File System, Memory, and Network, and by correlating events generated by multiple process executions.
B. Focus on *root* intrusions. Rather than trying to detect *all* possible intrusions, we focus only on root domain intrusions.
C. Separate detection of penetration and misbehavior. We employ two detectors that run in parallel: a *root penetration detector*, which looks for activities that exploit system vulnerabilities to penetrate the security perimeter, and a *root misbehavior detector*, which looks for misbehavior by root processes without trying to know the cause.
D. Design for real-time response with overhead control. We focus on making the intrusion response system real-time, but at the same time allow the system administrator to tune the impact to the foreground task response time distribution.

Several real-world exploits guided these design principles. We illustrate with two examples.

Example 2.1: To exploit a time-of-check to time-of-use (TOCTTOU) vulnerability [13], an attacker process performs two operations, *remove* and *replace*, between the check and use operations of the vulnerable process. The `rpm` package manager (runs as root) in Linux contains an `<open,open>` TOCTTOU pair [14]. An attacker process can replace the file created in the first `open` with a file of her choice, before it is `open`ed again for execution. Detecting such an exploit requires the detection of several events with associated data across two different processes.

Example 2.2: Rootkits typically modify system binaries (e.g., `ls, ps`, etc.) to conceal their presence [12]. For example, modification of the `/bin/ls` root-domain file can be interpreted as anomalous activity indicating presence of a rootkit. However, the package updaters (e.g., `yum`) may also modify system programs including `/bin/ls`, a benign event. Differentiating between these two anomalous activities requires context information and correlation of both *process* subsystem and *filesystem* activity. This knowledge must be combined with a mechanism that allows for exceptions when dealing with "apparently anomalous" events.

The justification of Principle A is straightforward and is illustrated by both the above examples. Detecting the exploit in Example 2.1 requires time-sensitive correlation of event data across two processes, while Example 2.2 necessitates correlation across events in the process subsystem (`exec` of yum) against a filesystem anomalous event (`write` of `/bin/ls`).

Principle B is probably more controversial; its justification involves considerations of both importance and feasibility. First, root intrusions are much more important than user-level intrusions—root intrusions can cause unlimited damage, while user-level intrusions cannot affect other users. Second, root intrusions are more feasible to detect than user-level intrusions, because root behavior is more constrained (and hence more predictable) than user behavior;[1] for instance, a root process modifying `/bin/ls` is in itself suspicious, unlike a user process modifying a user file. Further, the difficulty of accurately modeling "normal" user behavior could easily lead to an unacceptable false positive rate [15].

In addition, focusing on root intrusions is more feasible from the standpoint of efficiency in two respects – we reduce the analysis overhead of user-level events[2] and more importantly, we substantially reduce signature-matching overhead by eliminating signatures corresponding to the large population of user-level exploits.

Justification of Principle C relies on the observation that root domain intrusions can be detected in two distinct ways: (1) by observing activity that exploits known vulnerabilities to penetrate root protections, and (2) by simply observing misbehavior in root processes, even if the cause of the misbehavior is unknown. These two approaches are complementary: penetration detection is less prone to false positives, since it relies on specific signatures, while misbehavior detection is less prone to false negatives, since it can be effective even against intrusions that exploit unknown or unavoidable vulnerabilities, such as compromised root passwords. A novelty in Rootsense post-intrusion misbehavior detection lies in the ability to specify exceptions that help reduce false positives to address situations such as in Example 2.2.

Finally, justification of Principle D primarily harbors on practical usability of the intrusion prevention system. Information about and control over the response time distribution of foreground task response times can help the system administrator evaluate the suitability of such a solution in her current environment. The ability to trade-off average response-time for jitter (and vice-versa) is built into the design of Rootsense; this ability we believe is valuable in adapting Rootsense behavior in varied environments, such as a jitter-sensitive desktop environment or a throughput-sensitive server environment.

B. System Architecture

Rootsense uses a layered architecture composed of five distinct layers (illustrated in Figure 1). The *Monitored Subsystems* form the lowest layer and encapsulate the four main

[1]This detection domain can be expanded to include "non-human", privileged user accounts (e.g., apache, sshd, mailman, etc.) with constrained behaviors that can be easily modeled.

[2]To respond to privilege-escalating penetration attacks, however, some user event analysis is still required.

subsystems—Process, File System, Memory, and Network—that together compose the holistic view of the system. The *Sensors at Touchpoints* provide an abstraction layer on top of the detailed sensing mechanisms incorporated inside the monitored subsystems. The *Syscall Sensor* intercepts selected system calls as well as other kernel events along with associated data and forwards them for processing; third-party sensors can also be plugged into this layer (e.g., Snort events or a simple sensor for network bandwidth usage). The *Monitoring Modules* perform aggregation, filtering, and escalation functions on intercepted events before analysis.

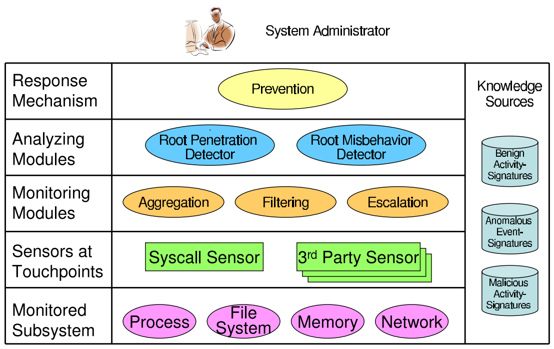

Fig. 1. Rootsense system architecture.

The next layer is composed of *Analyzing Modules* that use input from the monitoring modules and the *Knowledge Sources* to analyze the current state of the system. The following databases comprise the Knowledge Sources: (i) *Malicious Activity Signatures*, (ii) *Benign Activity Signatures*, and (iii) *Anomalous Event Signatures*. We describe these concepts in detail in Section III. Finally, in the topmost layer of this architecture is the *Response Mechanism*, which acts to stop an ongoing "detected" intrusion attempt and notifies the system administrator.

III. Concepts and Terminology

In this section, we introduce the basic concepts in Rootsense, including specific terminology that we shall use in the rest of the paper.

Subsystem	Sample events
Process	create process, process exit, load executable, signal process
Memory	allocate memory, free memory, map memory, unmap memory
Filesystem	open file, make directory, change permissions/owner, symlink, read/write
Network	create socket, listen, connect, close

TABLE I
Sample events for each subsystem.

Definition 3.1: An **event** is a *significant occurrence* within a single subsystem and due to a single process. An event description comprises key pieces of information relating to the occurrence including a unique event ID, it's invocation time, subsystem, process-id, user-id, effective user-id, event-type, and optional additional arguments.

Notation	Description
`lower-case letter`	Variable
`upper-case letter`	Predefined Set
`:X`	Member of X
`!(E)`	Not E
`*`	Don't-care

TABLE II
Notations in signature specification.

An event description in Rootsense is represented using the following format:

```
[event ID] [timestamp] [subsystem] [pid] [uid]
[euid] [event type] [arguments]
```

Event IDs are incremented for each new observed event. Table I lists example event types that may occur within each subsystem.

```
521 1122538450 P 25631 0 0 exec /usr/bin/yum 22
643 1246434546 F 25631 0 0 write /bin/ls 211
```

Fig. 2. An example of an **activity**

Definition 3.2: An **activity** is a sequence of events. The events in an activity need not occur consecutively; other events can be interleaved arbitrarily.

A useful activity monitored in Rootsense typically consists of a sequence of *correlated* events. An example of an activity is shown in Figure 2. Notice that the event IDs are not sequential, i.e., this is a filtered sample of the observed system events. This activity consists of two events:

Event 521. At time 1122538450, process with PID 25631 which is running with both real and effective UID of root (0) invokes a *Process* subsystem *Exec* event to load the `/usr/bin/yum` executable, and

Event 643. At time 1246344546, the above process (same PID), causes a *filesystem* event of type *Write*; the file written to is `/bin/ls`.

Definition 3.3: An **event signature** is a generic event which does not include timestamp information and allows variables and wild-cards (don't-cares) in various event description fields.

In other words, an event signature is not a system occurrence but is used to represent a class of actual system events, allowing for system variants such as their time of occurrence, associated process ID, etc.

Table II lists the notations for variables and wild-cards that are used in Rootsense for specifying event signatures. An example of an event signature is shown below:

```
P p u 0 exec :U * *
```

The above event-signature represents the class of actual events which occur in the *process* subsystem (P), caused by a process with *effective user ID (EUID) root (0)*, with the event-type *exec*, and wherein the executable loaded by exec is a member of the externally-specified set *U* of executables. The process ID (p) and the user ID (u) are specified as variables, and the

```
<begin signature>
<extern UpdateSet:U>
<extern ProtectedSet:P>
P p u 0 exec :U * *
F p u 0 write :P * *
<end signature>
```

Fig. 3. An activity signature corresponding for the specific activity in Figure 2.

arguments of exec are "don't-cares". We eliminate the event ID and timestamp for brevity of the event signatures.

Definition 3.4: An **activity signature** is a sequence of event signatures and describes classes of activities.

Figure 3 is an activity signature that captures the behavior of the activity described in Figure 2. The activity signature first specifies that there are two externally defined sets, *U* and *P*, that are used in the activity signature. Next, the main signature body is specified and it consists of two event signatures:

Event Signature 1 requires that a matching event be a *process* subsystem *exec* event, invoked by a process with EUID 0 and executable loaded is an element of the UpdateSet, *U*. The remaining arguments to exec are "don't-cares". The process ID (p) and the user ID (u) are specified as variables, implying that although they are not specified, their values are significant connecting information for subsequent *events* that contribute to an *activity signature* match.

Event Signature 2 requires a matching event to be a *filesystem write* event to a file in the ProtectedSet *P*. The PID (p) and the UID (u) must exactly match those of the event matching event signature 1. The remaining arguments are "don't-cares".

A subtlety in this signature is that we want the *same* process p to participate in all three events. But it is conceivable that the process with PID p might exit and that another process with the same PID p could later be created. Our implementation deals with this by aborting the matching of any activity signatures waiting for an event involving a process with PID p if that process exits.

The activity signatures are totally ordered. To model situations in which the order of some events is irrelevant, we simply use multiple activity signatures to describe all allowable orderings in our prototype implementation. Of course, this is much less concise than a partially-ordered, graph representation [1].

Rootsense uses three kinds of signatures within its penetration and misbehavior detectors. *Malicious activity signatures* are intended to capture the behavior of a system while a vulnerability is being exploited, allowing the detection of intrusions "at entry". *Anomalous event signatures* describe single root events that are apparently malicious; these are used to detect intrusions "post entry". An example is a file in the `/bin/*` ProtectedSet being written to. Finally, *benign activity signatures* are intended to describe scenarios in which apparently malicious events are actually innocent. (Example 2.2 describes such a scenario.) As indicated in Figure 1, Rootsense uses three databases, one with each of the above kinds of signatures. We explain how Rootsense uses these databases to detect intrusions next.

IV. Intrusion Detection and Response

In this section, we describe the dual approach to intrusion detection in Rootsense, addressing the challenges associated with maintaining and updating the signatures databases. We follow this with a description of the real-time intrusion response mechanism and also describe how a system administrator can control Rootsense overhead to better match system end-goals.

A. Dual Detection in Rootsense

Rootsense employs a dual approach to intrusion detection, using both a *root penetration detector* as well as a *root misbehavior detector*. The penetration detector looks for activities that exploit system vulnerabilities to penetrate the security perimeter, while the misbehavior detector looks for activities by root processes that constitute misbehavior. Both detection engines run continuously, in parallel.

For monitoring the global system state continuously, each detector independently monitors and correlates events relevant to its operation. The detectors do not correlate and track all activities; they use their associated signatures databases to determine important activities that must be tracked. By observing the system activity in real time, each detector tracks relevant transitions in the global system state to make inferences about the system's intrusion status.

Root Penetration Detector. The *root penetration detector* monitors all system activities to monitor the "entry points" for intrusions, which are the system vulnerabilities. It continuously monitors all events for matches with activity signatures in the *malicious signatures database*; a match indicates an intrusion attempt and triggers the response mechanism.

Although the root penetration detector is intended to detect root domain intrusions only, it still must monitor all activities by both users and root. This is required because several intrusions of the root domain are due to privilege escalating exploits (e.g., a TOCTTOU vulnerability exploit) wherein a user process acts maliciously to obtain root privilege. To track important activity in the system, the detection engine performs the following actions for each event observed:

1. For each malicious activity signature whose *first* event signature matches the observed event, create an *activity state machine* which will track the specific activity pattern. The state machine is populated with event-specific information to replace a subset of the variables in the activity signature. The state machine is then set to *wait-for* an event that matches the *second* event signature in the activity signature.

2. For each state machine whose *wait-for* event signature matches the observed event, transition the state machine to the next state by setting it to wait-for an event that matches the *next* event signature in the activity signature. Before doing so, populate the state machine with event-specific information to replace any additional variables it may have instantiated. If a state machine terminates due to this event, invoke the response mechanism

3. If the event belongs to the *terminating* class of events, abort all of the affected state machines. For example, with a state machine whose *wait-for* event is an event with process with PID 16909, termination of that process will disable the state machine.

The root penetration detector cannot detect all root penetrations. Specifically, it cannot detect intrusions in the following three scenarios: (i) an unknown system vulnerability is exploited, (ii) a known vulnerability is exploited, but it is not possible to detect the exploit with within the framework of Rootsense, and (iii) the intruder makes a "legal" entry into the system, using a compromised root password. Detecting such intrusions is the job of the *root misbehavior detector*, which we describe next.

Root Misbehavior Detector. The *root misbehavior detector* monitors only the activity of processes with effective user ID (EUID) root to detect privileged-mode misbehavior. It works by looking for single events that match the *anomalous event signatures* database. Recall that such events are apparently malicious. However, there may be scenarios in which such events are actually innocent; to recognize those situations, we must simultaneously track matches within the *benign activity signatures* database. More precisely, the root misbehavior detector processes each event that it observes by

- creating and/or advancing state machines (i.e., do steps 1 and 2 of the *root penetration detector* operation) for any of the benign activity signatures that can make progress on this event, and
- simultaneously checking whether the event matches some anomalous event signature.

If the event matches an anomalous event signature, then we regard it as malicious and invoke the response mechanism *unless* at least one benign activity state machine was able to make progress on the event, in which case we judge the event to be benign.

One might wonder about a situation in which a benign activity signature is *partially* matched, but never completed. Might this cause an anomalous event to be wrongly judged benign? We actually believe that this situation will never arise, because we conjecture that it suffices to use benign activity signatures containing just *one* anomalous event signature, which always occurs *last*. If this is so, then whenever a benign activity signature "salvages" an apparently malicious event, that actually completes the match of the benign activity signature.

Signatures Databases. One of the key challenges in Rootsense is the generation and continuous updating of the signatures databases. To make the task of signature specification more efficient and practical, our approach builds *generic* classes of malicious, anomalous, and benign activity. To keep the signature count tractable, we classify both subjects as well as objects using the set concept to aggregate specific vulnerabilities into classes. Set information associated with an activity signature is specified as *external* information that can potentially be shared by several activity signatures.

```
<begin signature>
<extern CheckSet:C>
<extern RemoveSet:R>
<extern CreationSet:L>
<extern UseSet:U>
F p u 0 :C f *
F !p * !0 :R f *
F !p * !0 :L f *
F p u 0 :U f *
<end signature>
```

Fig. 4. An **activity-signature** corresponding to the TOCTTOU serialization vulnerability class [16].

The current Rootsense prototype consists of hand-coded signature sets for modeling malicious activity. To obtain malicious activity-signatures, we started with the combined findings of classification proposals for software vulnerabilities by Neumann [17], Landwehr et al. [16], and Bishop [18]. We analyzed each vulnerability class to determine if generic exploits could be specified independent of context and associated data. The identified exploits were then encoded into signatures interpreted by Rootsense. For instance, a generic exploit signature for the serialization TOCTTOU vulnerability (as classified in [16]) is specified in Figure 4. Set specification is borrowed from the recent work on understanding TOCTTOU vulnerabilities in file systems by Wei et al. [14]

The anomalous event signatures database is a collection of event signatures for EUID root processes, each of which by itself constitutes a potential anomaly in root behavior. These event signatures are specified manually, and include events such as a process with effective UID root writing to /bin/* or exec'ing a shell.

An example of a benign activity signature was introduced earlier in Figure 3. While these are manually specified in our prototype, in the future, however, we envision obtaining this class of activity signatures automatically. This can be accomplished by first running the system in a protected environment guaranteed to be free of malicious activity (similar to the techniques used in studies such as [19] for learning normal program behavior) and monitoring for anomalous events. An automatic process for determining correlated events to form the activity and another process for deriving a generic signature that considers activity invariants might then be used.

Efficient Signature Matching in Rootsense. In Rootsense, the events are analyzed in parallel by the *Root Misbehavior Detector* and the *Root Penetration Detector*. The detection engines employ the following hash-table data structures to speedup the pattern matching process: (i) a **signatures hash-table**, for fast look-up of *signatures* (value) whose first events conform to the <*subsystem,event-type*> (key) of the observed event, (ii) a **state machines hash-table**, for fast look-up of *state machines* (value) whose wait-for events conform to the <subsystem,event-type> (key) of the observed event, and (iii) an **anomalous event signatures hash-table**, for fast look-up of *event signatures* (value) conforming to the *(subsystem,event-type)* (key) of the observed event. The termination status of the state machine determines the state of

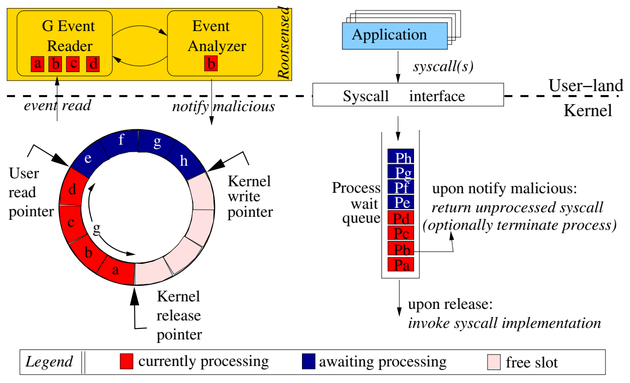

Fig. 5. Rootsense kernel-space sensing/response and user-land interactions.

the system and whether or not the response mechanism is invoked.

B. *Intrusion Response*

Timely response to intrusion attempts, before they can perpetrate damage, is necessary to ensure system security. Timely response in Rootsense is ensured by analyzing each system call request, before the actual system call is invoked. Doing so naively, however, can introduce unnecessary context switching between the Rootsense kernel space (i.e., sensing/response) and user space (i.e., monitoring/analysis) components. Consequently, a naive implementation can incur an arbitrary amount of system overhead.

Timeliness of Response. Figure 5 depicts the sensing and response mechanisms in Rootsense, including the interactions between the kernel and user space components. The Rootsense kernel space components (sensing and response) interpose between the system call interface and the call implementation.

When an application invokes a system call, this system call event is assigned a unique *event ID* and buffered in a circular *kernel event buffer* for intrusion analysis; the process is put to sleep in the *process wait queue*. The Rootsense monitoring component is implemented as a *G event reader* which consumes from the event buffer with a *G* event granularity. The *G* events are then analyzed, followed by consuming of the next *G* events from the kernel event buffer. Rootsense's intrusion response mechanism is invoked each time the analyzer concludes that a requested system event is potentially malicious. The analyzer notifies the response mechanism the associated *event ID* of the potentially malicious event, which is used to wake up the corresponding process from the process wait queue. This process is then either *terminated* or its system call request is denied, depending on the administrator specified policy for handling suspected malicious events, and the system administrator is notified. Finally, each time the user space component consumes a new set of *G* events, processes associated with the previous *G* event IDs from the process wait queue are woken up and allowed to continue forth executing their respective system calls, since those events are implicitly identified as non-malicious by the response mechanism.

Controlling Overhead Characteristics. Intrusion detection and prevention systems necessarily introduce system overhead because they must analyze system activity and such analysis requires system resources. While optimized implementations can reduce the absolute value of such overhead, the characteristics of such overhead are often hard to control. The characteristics of system overhead can be captured by the distribution of foreground task response times, which can be succinctly represented by *average* and *standard deviation* values. While the average value dictates overall system interactivity, the standard deviation (i.e., variability in response times) dictates user-perceived jitter. In interactive systems, minimizing jitter may be as important as improving average interactive performance.

Intrusion analysis using a *G* event granularity was a design choice in Rootsense to control the number of context switches between user and kernel space as well as the context switches between the Rootsense user space and the system foreground tasks (i.e. applications). The value of *G* is a control parameter that a system administrator can use to trade-off average-case interactive performance with jitter. Increasing *G* implies that the Rootsense user space process can analyze a larger number of events before issuing a system call to consume subsequent events (i.e., requiring a user-kernel context switch). It also implies that foreground tasks have an increased chance of waiting *after*, rather than waiting both *before* and *after*, they enqueue their respective events, thereby reducing context-switch overhead introduced into foreground workload. Thus, a larger value of *G* improves interactivity, but also increases jitter. Interestingly, the size of the kernel event buffer does not impact overhead characteristics, as long as its value is greater than *G*. This can be intuitively reasoned by considering that in steady state, when the rate of event analysis is greater than the rate of event generation, process wait events are controlled by event consumption behavior which is independent of kernel event buffer size. We validated this intuition experimentally and report results in Section V.

Rootsense exports tunability of *G* to the system administrator who can observe as well as control this trade-off based on the needs of her system. For instance, the administrator at a data-center hosting web-services may choose to exclusively prioritize average-case response time, while a system administrator for an end-user system may choose to prioritize jitter reduction to keep user experience acceptable.

V. Evaluation

We deployed Rootsense on an Intel x86 2GHz machine with 512MB of memory, running GNU/Linux with a Fedora Core 5 distribution and 2.6.15.6 kernel.[3]

The kernel component overwrites the `sys_call_table` entries for interposition, and exports events via a `/proc` extension. Rootsense was evaluated for both accuracy and performance, as we shall describe next.

[3]The only exception was the `execve/ptrace` exploit for which we used a 2.2 kernel.

Vulnerability	Symptom Description	Exploit-based	Anomaly-based	Rootsense
bzip2 [20]	`bzip2 chmod`, permissions, race condition	$- +$	$\ominus +$	$- +$
execve [21]	`execve/ptrace`, race condition	$- +$	$\ominus +$	$- +$
tar [22]	GNU `tar` hostile destination path	$- +$	$\ominus +$	$- +$
tOrnkit [23]	Malicious modification of `/bin/ls`	$\ominus +$	$- +$	$- +$
X.org [24]	`Xorg` X window server local privilege escalation	$\ominus +$	$- +$	$- +$
N/A	Benign modification of `/bin/ls`	$- +$	$- \oplus$	$- +$
N/A	Benign /bin/sh exec by root `/bin/bash` process	$- +$	$- \oplus$	$- +$

TABLE III
COMPARING ROOTSENSE ACCURACY WITH EXPLOIT-BASED AND ANOMALY-BASED DETECTORS.

Rootsense Accuracy. To evaluate accuracy, we used a unpatched Fedora Core 5 distribution as a test-bed, containing a few *root vulnerabilities*. We installed additional vulnerable root binaries to complete the test-bed. Next, exploits corresponding to these vulnerabilities were carried out to determine the Rootsense response. In addition, we simulated apparently anomalous behaviors by benign processes.

The objective of our evaluation was not to evaluate detection completeness, but rather to demonstrate the scope of Rootsense's capability to distinguish between malicious and non-malicious activity. It is indeed infeasible to evaluate completeness, since the duel between intruders and intrusion prevention systems is necessarily never-ending. Consequently, we restrict this evaluation to merely contrasting the behavior or Rootsense in comparison with generic exploit-based and anomaly-based systems alone.

Table III presents a comparative evaluation of exploit-based, anomaly-based, and Rootsense detection accuracy. Accuracy is represented by true (-/+) as well as false ($\ominus$/$\oplus$) positives/negatives, respectively. For brevity, we only elaborate three specific scenarios from Table III, the `bzip2` race attack, the the `Xorg` local privilege escalation, and the benign modification of `/bin/ls`.

The `bzip2` race attack [20] has a clearly defined exploit-signature, requiring the attacking process to interpose an `<unlink,link>` pair between `open` and `chown` operations of bzip2. Both the exploit-based detector and Rootsense are successful in preventing this attack. However, since this attack does not involve any anomalous behavior by the `bzip2` process, the anomaly-based detector makes a false negative decision. The `Xorg` privilege escalating attack exploits a buffer overflow. It is well-known that this class of attack cannot be detected "at entry" without architectural changes or binary modification; consequently, exploit-based detectors make a false negative decision. However, post-intrusion, this attack `execs` a shell, inheriting the privilege of `Xorg` (i.e., root). An anomaly-based detector which tracks shell exec'ing behavior (including Rootsense) can detect and prevent this immediately after the intrusion. Finally, the benign modification of the protected binary `/bin/ls` by `yum` results in false positive decision by the anomaly-based system, but Rootsense correctly identifies the activity as benign. This capability of coding exceptions to anomalous behavior in Rootsense is a unique one which enables combining exploit-based and anomaly-based detection without sacrificing accuracy.

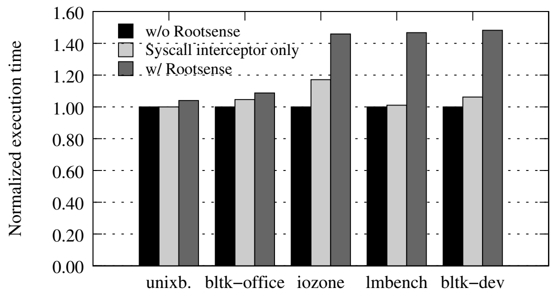

Fig. 6. Benchmark execution times.

Rootsense Performance. In evaluating Rootsense performance, we focused on three aspects: (a) system overhead, (b) impact and control over foreground response time distribution, and (c) effect of knowledge base size. Unless otherwise mentioned, the knowledge base contained approximately ten each of malicious activity signatures, benign activity signatures, anomalous event signatures, and set definitions.

First, to evaluate system overhead, we ran the several benchmarks on three different system configurations - a vanilla system, a system with Rootsense kernel space component but a dummy user space component (i.e., no event correlation and analysis), and Rootsense (everything enabled). For each benchmark, we present a brief description next along with the load it generates by indicating its measured event generation rate in parentheses (events per second). Lmbench (11575) is a suite of diverse micro-benchmarks used to measure operating system performance; it issues various system calls at high rate. UnixBench (784) is a set of micro-benchmarks which range from system to arithmetic related. Iozone (2468) is an I/O intensive benchmark with mostly disk reads and writes. Bltk-dev (2910) and Bltk-office (1232) are end-user workloads that real world usage such as software development and office related applications, respectively.

Figure 6 depicts the normalized execution times of the benchmarks with the three configurations. With Rootsense, the execution time increases by as much as 4% with loads such as Unixbench and Bltk-office, and as much as 45% with the Lmbench (the highest load) benchmark. This indicates that the system can slowdown as much as 45% for small knowledge base sizes. For most benchmarks, the interceptor component of Rootsense incurs a small amount of overhead; the bulk of the overhead resides in the analysis phase.

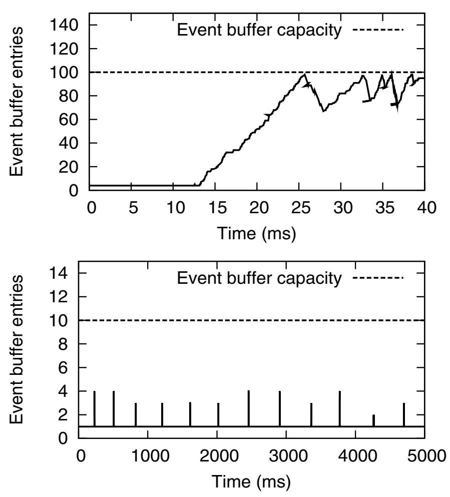

Fig. 7. Event buffer entries with high load (above) and low load (below).

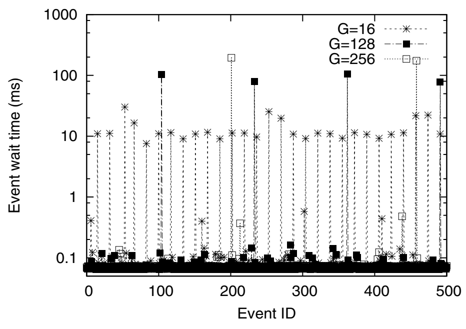

Fig. 8. Event wait time characteristics.

To study controllability of response time distribution, we used two system variables - the kernel *event buffer size* (K) and the user-space *event processing granularity* (G). Figure 7 shows how the number of entries in the event buffer varies over time in the case of high-load and low-load. With high load (upper graph), foreground tasks must wait for events to be processed by Rootsense before generating more events; consequently, the event buffer is mostly close to full in the steady state. With low load (lower graph), Rootsense ends up waiting for events to be produced and the event buffer always runs low. We also studied the effect of varying the kernel event buffer size and found that it did not affect the execution time of any benchmark (we elaborate on this later in Figure 9.)

Next, we evaluated the impact to foreground response time distribution by measuring the *event wait time distribution* introduced by the Rootsense kernel-space component. This distribution directly measures the impact of Rootsense to foreground task performance, since it gets added to the vanilla response time distribution of any foreground workload. Figure 8 depicts the effect of varying the event processing granularity (G) on the event wait times when running the Lmbench benchmark. (Y-axis is shown in logarithmic scale). For a low value of G=16, most event wait times are distributed in the 10-20 ms range, with low variance. For a moderate G=128, there is a larger variance with most values lying in the 0.1-0.2 ms range, but a few that require up to 100 ms. This implies relatively larger jitter but lower average wait time. For a high G=256, this variance is even larger but with even fewer outliers . Thus, exporting control over the value of G to the system administrator enables fine-grained control over foreground response time distribution.

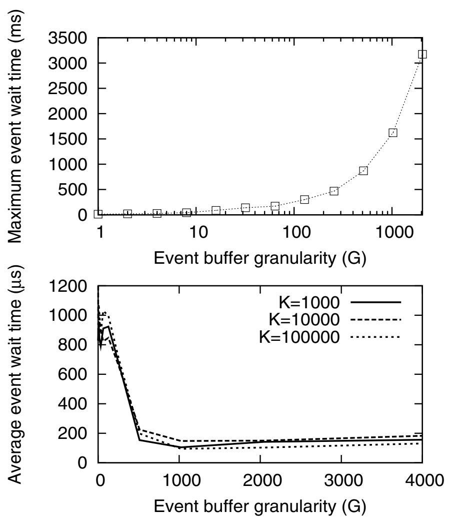

Fig. 9. Event wait times: maximum [K=10000] (above) and average (below).

This phenomenon is further elaborated in Figure 9 where we depict maximum event wait time as G is varied (upper graph) and show that average wait time is mostly independent of the event buffer size (K). The maximum event wait times increase proportionately to the event processing granularity, G, as can be seen in the upper graph (note that X-axis uses logarithmic scale), implying that a smaller G can reduce the jitter experienced by foreground tasks, confirming earlier deductions. Another noteworthy point (lower graph) is that smaller values of G can degrade average case performance almost exponentially. Values of G above 500 work well for all values of K.

Finally, to evaluate the impact of the size of the knowledge base, we populated the knowledge-base by generating random, but valid, event and activity signatures as required. We chose an equal mix of of *malicious activity*, *benign activity*, and *anomalous event* signatures. Figure 10 depicts how Lmbench execution time varies as the *number of signatures* and *states-per-signature* are scaled up. There is a relatively small increase in execution time overall – less than 10% relative to the 1 signature / 1 states-per-signature case. We can also conclude that the overhead depends mostly on the absolute number of signatures, rather than the states-per-signature.

VI. Related Work

While there exists a large body of work in this space, we only examine those closely related to Rootsense.

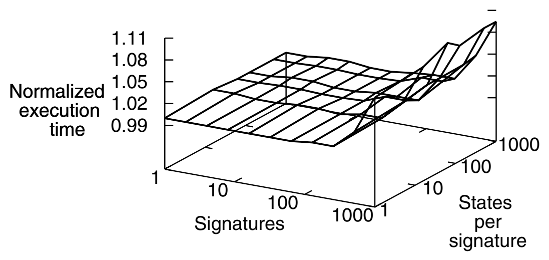

Fig. 10. Normalized Lmbench times when varying the knowledge base size.

a) Specification/Policy Based Approaches: Early work by Ko *et al.* [25] proposed specification-based detection that relies on specifications of intended program behavior, allowing the detection of unknown attacks. Systrace is a system call policy enforcement framework for restricting the usage of kernel services [26]. Minix3, a microkernel-based operating system, also implements system call usage policies for (traditionally in-kernel) user-space Minix3 services [27]. Similar high-level specifications for restricting program behavior are used in [2]. Manually specifying the behavior of each program is time-consuming and capturing all possible execution sequences in a multi-threaded, multi-programmed environment is infeasible. Rootsense attempts to address efficiency, effectiveness, and practicality with generic signature-sets that model both vulnerability exploits as well as misbehavior, completely bypassing application behavior specification. Rootsense also provides the additional flexibility that policy exceptions can be made by specifying appropriate benign activity signatures.

b) System Monitoring and Behavior Modeling: System call interposition for system monitoring as well as modeling the control flow of programs has been used widely since the early work by Forrest *et al.* [19]. Sekar *et al.* [2] proposed intrusion prevention systems that model and enforce correct program behavior through system call interception at runtime. However, work by Tan *et al.* [28] and Kang *et al.* [29] showed that malicious attacks can be evaded with sequences of legitimate system calls. This happens because the systems investigated did not consider system calls from all their perspectives, usually ignoring the arguments. Mutz *et al.* [30] propose monitoring individual applications to derive normal behavior in terms of both the control flow of system calls and the data flow in terms of the arguments supplied and return values obtained, rather than simply using system call sequences. In contrast, we suggest monitoring of an entire system (in uncompromised) state (rather than each application individually) to derive benign activity signatures corresponding to specific anomalous event signatures, to reduce false positives.

Gao *et al.* [31] propose detecting compromised programs by comparing the *behavioral distance* of distinct implementations of the program on the same inputs. Giffin *et al.* [32] also model a program's behavior with additional contextual information such as configuration files, command-line arguments, and environment variables. Similar to the above contributions, Rootsense uses control flow as well as data-flow as a basis for detection. However, in contrast to the application-specific nature of the above, Rootsense allows capturing system-wide behavior of multiple interacting programs.

c) Exploit-Based Penetration Detection: Traditionally, intrusion detection techniques have detected either vulnerability-exploiting penetration attacks or post-intrusion anomalous program behavior. Porras *et al.* have proposed STAT [1], a state transition analysis tool that models and detects penetrations as a series of state changes. The *root penetration detector* in Rootsense performs a similar function. In addition, Rootsense employs a *root misbehavior detector* for detecting post-intrusion anomalous behavior and proposes a method whereby the detectors can improve each other. Recent work on vulnerability-specific predicates proposed by Joshi *et al.* [3] allows detecting the exploitation of specific known vulnerabilities in software both in the past as well as present. In contrast, the signatures used in Rootsense are generic, but also provide for describing specific sets of objects that possess similar characteristics.

d) Real-time Intrusion Detection and Response: Somayaji *et al.* [33] created a tool that can stop attacks before they occur by delaying anomalous system calls possibly until they are completely stopped. This means that false positives only delay the execution of the malicious system call. Although Rootsense focuses on reducing the number of false positives, the previous approach is complementary and can be easily used in Rootsense. Lee *et al.* [34] argued that there exists a significant trade-off between performance and detection accuracy. Moreover, the authors argue the need of designing this trade-off in a adaptive manner. In contrast, we argue that performance and accuracy can be made independent, moving the trade-off to the response time distribution.

e) Others: Data mining techniques, with the ability to discover consistent and useful patterns of system features as well as describing program and user behavior have been used for intrusion detection [35]. More recently, Wang *et al.* [36] argue for holistic approaches that are able to correlate alerts during a multi-step network intrusion. Anagnostakis *et al.* [37] propose post-processing of anomalous behavior data with shadow honeypots to reduce false positives. We believe that signature generation and upkeep in Rootsense could benefit by using the techniques presented in these contributions.

VII. Conclusions and Future Work

Rootsense is a holistic approach for detecting and preventing host intrusion attempts in real-time. The four design principles of Rootsense help towards making it an accurate, effective, and practical intrusion prevention solution. Notably, Rootsense monitors global system state, employs dual detection of root domain intrusions, reduces false positives by allowing anomaly exceptions, and allows for fine-grained control over the impact to the foreground task response-time distribution. Experiments with a Linux-based Rootsense implementation, using several real-world exploits and a range of end-host and server performance benchmarks, show that Rootsense can accurately prevent a range of real-world attacks with an acceptable level of *controllable* system overhead.

We see several avenues for future work and describe two that we are actively pursuing. First, the current prototype uses a simple language for specifying program behavior; activities involving more than one possible sequence of operations are modeled using multiple signatures. We are exploring language frameworks similar to STATL [38] that provide richer capability for modeling system activity. Second, when an intrusion is observed by the root misbehavior detector, the intrusion can be backtracked using the technique proposed by King et al. [39] to obtain the penetration activity sequence automatically. The challenge then is to derive a new malicious activity signature from the specific activity automatically. The root penetration detector would then use this new signature to monitor a previously unknown vulnerability class.

Acknowledgments

We thank the anonymous reviewers for their detailed feedback which helped improve the paper. This work was supported by the National Science Foundation under grants IIS-0552555, IIS-0534530, and the Department of Energy grant DE-FG02-06ER25739. However, the opinions and findings presented in this paper do not necessarily reflect the views of the above agencies.

References

[1] K. Ilgun, R. A. Kemmerer, , and P. A. Porras, “State Transition Analysis: A Rule-Based Intrusion Detection Approach,” *IEEE Transactions on Sortware Engineering*, vol. 20, no. 5, March 1995.

[2] R. Sekar and P. Uppuluri, “Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications,” *Proc. of the USENIX Security Symposium*, August 1999.

[3] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen, “Detecting Past and Present Intrusions through Vulnerability-Specific Predicates,” *Proc. of the Symposium on Operating Systems Principles*, October 2005.

[4] D. Denning, “An Intrusion-Detection Model,” *IEEE Transactions on Software Engineering*, vol. 13, no. 2, pp. 222–232, February 1987.

[5] S. A. Hofmeyr, S. Forrest, and A. Somayaji, “Intrusion Detection Using Sequences of System Calls,” *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998.

[6] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, “A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors,” *Proc. of the IEEE Symposium on Security and Privacy*, May 2001.

[7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, “StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks,” *Proc. of the 7th USENIX Security Conference*, January 1998.

[8] G. H. Kim and E. H. Spafford, “The Design and Implementation of Tripwire: A File System Integrity Checker,” in *Proc. of the ACM Conference on Computer and Communications Security*, 1994.

[9] M. Roesch, “Snort - Lightweight Intrusion Detection for Networks,” *Proc. of the Large Systems Administration Conference*, November 1999.

[10] V. Paxson, “Bro: A System for Detecting Network Intruders in Real-Time,” *Proc. of the USENIX Security Symposium*, January 1998.

[11] C. Ko, G. Fink, and K. Levitt, “Automated Detection of Vulnerabilities in Priviledged Programs by Execution Monitoring,” *Proc. of the 10th Annual Computer Security Applications Conference*, December 1994.

[12] K. M. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Shermann, and K. A. Oostendorp, “Confining Root Programs with Domain and Type Enforcement (DTE),” *Proc. of the USENIX Security Symposium*, July 1996.

[13] M. Bishop and M. Dilger, “Checking for Race Conditions in File Accesses,” *Computing Systems*, vol. 9, no. 2, pp. 131–152, 1996.

[14] J. Wei and C. Pu, “TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study,” *Proc. of the USENIX Conference on File and Storage Technologies*, December 2005.

[15] S. Axelsson, “The Base-Rate Fallacy and the Difficulty of Intrusion Detection,” *ACM Transactions on Information System Security*, vol. 3, no. 3, pp. 186–205, Aug. 2000.

[16] C. E. Landwehr, A. R. Bull, J. P. McDermott, and W. S. Choi, “A Taxonomy of Computer Program Security Flaws,” *ACM Computing Surveys*, vol. 26, no. 3, pp. 211–254, September 1994.

[17] P. G. Neumann, “Computer System Security Evaluation,” *Proc. of the National Computer Conference*, June 1978.

[18] M. Bishop, “A Taxonomy of Unix System and Network Vulnerabilities,” *University of California, Davis Technical Report CSE-9510*, May 1995.

[19] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, “A Sense of Self for Unix Processes,” in *Proc. of the IEEE Symposium on Research in Security and Privacy*, May 1996.

[20] Security Focus, “BZip2 CHMod File Permission Modification Race Condition Weakness,” *http://www.securityfocus.com/bid/12954/*, 2005.

[21] Wojciech Purczynski / cliph / ¡wp@elzabsoft.pl, “Exploit for execve/ptrace race condition in Linux kernel up to 2.2.19,” , 2001.

[22] Security Focus, “GNU Tar Hostile Destination Path Variant Vulnerability,” *http://www.securityfocus.com/bid/5834/*, 2002.

[23] F-Secure, “F-Secure Virus Descriptions : Tornkit,” *http://www.f-secure.com/v-descs/torn.shtml*, 2001.

[24] Security Focus, “X.Org X Window Server Local Privilege Escalation Vulnerability,” *http://www.securityfocus.com/bid/17169/*, 2006.

[25] C. Ko, M. Ruschitzka, and K. Levitt, “Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach,” *Proc. of the IEEE Symposium on Security and Privacy*, May 1997.

[26] N. Provos, “Improving Host Security with System Call Policies,” *Proc. of the USENIX Security Symposium*, August 2003.

[27] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, “Modular System Programming in MINIX 3,” *;login: The USENIX Magazine*, vol. 31, no. 2, April 2006.

[28] K. M. C. Tan and R. A. Maxion, “”why 6?” defining the operational limits of stide, an anomaly-based intrusion detector,” in *Proc. of the IEEE Symposium on Security and Privacy*, 2002.

[29] D.-K. Kang, D. Fuller, and V. Honavar, “Learning classifiers for misuse and anomaly detection using a bag of system calls representation.” in *Proc. of 6th IEEE Systems Man and Cybernetics Information Assurance Workshop (IAW).*, 2005.

[30] D. Mutz, F. Valeur, C. Kruegel, and G. Vigna, “Anomalous System Call Detection,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 9, no. 1, pp. 61–93, February 2006.

[31] D. Gao, M. K. Reiter, and D. Song, “Behavioral Distance for Intrusion Detection,” in *Proc. of RAID 2005*, September 2006.

[32] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller, “Environment-Sensitive Intrusion Detection,” *Proc. of the International Symposium on Recent Advances in Intrusion Detection*, September 2005.

[33] A. Somayaji and S. Forrest, “Automated response using System-Call delays,” in *Proc. of the USENIX Security Symposium*, August 2000.

[34] W. Lee, J. B. D. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang, “Performance Adaptation in Real-Time Intrusion Detection Systems,” in *Recent Advances in Intrusion Detection*, 2002.

[35] W. Lee and S. J. Stolfo, “A Framework for Constructing Features and Models for Intrusion Detection Systems,” *ACM Transactions on Information and Systems Security*, vol. 3, no. 4, pp. 227–261, 2000.

[36] L. Wang, A. Liu, and S. Jajodia, “An Efficient and Unified Approach to Correlating, Hypothesizing, and Predicting Network Intrusion Alerts,” *Proc. of the 10th European Symposium on Research in Computer Security (ESORICS 2005)*, September 2005.

[37] K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. Keromytis, “Detecting Targeted Attacks Using Shadow Honeypots,” *Proc. of the USENIX Security Symposium*, August 2005.

[38] S. T. Eckmann, G. Vigna, and R. A. Kemmerer, “STATL: An Attack Language for State-based Intrusion Detection,” *Journal of Computer Security*, vol. 10, no. 1/2, pp. 71–104, 2002.

[39] S. T. King and P. M. Chen, “Backtracking Intrusions,” *Proc. of the Symposium on Operating Systems Principles (SOSP)*, October 2003.