Exploiting Execution Context for the Detection of Anomalous System Calls

Darren Mutz, William Robertson, Giovanni Vigna, and Richard Kemmerer

Computer Security Group
Department of Computer Science
University of California, Santa Barbara
{dhm,wkr,vigna,kemm}@cs.ucsb.edu

Abstract. Attacks against privileged applications can be detected by analyzing the stream of system calls issued during process execution. In the last few years, several approaches have been proposed to detect anomalous system calls. These approaches are mostly based on modeling acceptable system call sequences. Unfortunately, the techniques proposed so far are either vulnerable to certain evasion attacks or are too expensive to be practical. This paper presents a novel approach to the analysis of system calls that uses a composition of dynamic analysis and learning techniques to characterize anomalous system call invocations in terms of both the invocation context and the parameters passed to the system calls. Our technique provides a more precise detection model with respect to solutions proposed previously, and, in addition, it is able to detect data modification attacks, which cannot be detected using only system call sequence analysis.
Keywords: Intrusion Detection, System Call Argument Analysis, Execution Context

1 Introduction

A recent thrust of intrusion detection research has considered model-based detection of attacks at the application level. Model-based systems operate by comparing the observed behavior of an application to *models* of normal behavior, which may be derived automatically via static analysis [8, 23] or learned by analyzing the run-time behavior of applications [3, 5, 12, 18, 15]. In each case, attacks are detected when observed behavior diverges in some respect from the normal behavior captured by the model. In contrast to misuse-based approaches, where the analysis identifies attacks against applications using patterns of known malicious actions, model-based schemes have the advantage of being able to detect novel attacks, since attacks are not explicitly represented by the system. We note that this advantage typically comes at the cost of performance, precision, and explanatory capability, three properties that misuse-based approaches often achieve very well.

Most model-based intrusion detection systems monitor the sequence of system calls issued by an application, possibly taking into account some execution state. For example, the system described in [3] monitors pairs of system calls and records the application's stack configuration (that is, part of the history of function invocations). During the detection phase, the system checks if the observed pairs of system calls (and their associated stack configuration) match pairs recorded during the learning period. The systems described in [8] and [23] check call sequences against automata-based models derived from the application's source code or binary representation, and identify sequences that could not have been generated by the model.

Some of the shortcomings of sequence-based approaches were discussed in [2], where the problems of *incomplete sensitivity* and *incomplete sets of events* were introduced. Incomplete sensitivity affects models derived from static analysis. Due to the limitations of static analysis techniques, these models may accept impossible sequences of system calls (for example because branch predicates are not considered).

The problem of incomplete sets of events is more general, and it affects all approaches based on system call sequences. This problem stems from the fact that, in these systems, the manifestation of an attack must be characterized in terms of anomalies in the order in which system calls are executed. Changes in the ordering of system call invocations occur, for example, because foreign code is injected into the application (such as through a buffer overflow) or because the order in which instructions are executed is modified. Therefore, by modeling system call sequences, these approaches implicitly restrict themselves to only detecting attacks that modify the execution order as expressed by the application's code or by the execution histories observed during a training period. Unfortunately, an attacker can successfully compromise an application's goals by modifying the application's *data* without introducing anomalous paths in the application's execution.

```c
void write_user_data(void)
{
  FILE *fp;
  char user_filename[256];
  char user_data[256];

  gets(user_filename);

  if (privileged_file(user_filename)) {
    fprintf(stderr, "Illegal␣filename.␣Exiting.\n");
    exit(1);
  }
  else {
    gets(user_data);      // overflow into user_filename

    fp = fopen(user_filename , "w");

    if (fp) {
      fprintf(fp, "%s", user_data);
      fclose(fp);
    }
  }
}
```

Fig. 1. Sample data modification attack.

Consider, for example, the procedure `write_user_data` in Figure 1. Here, an overflow of the variable `user_data` at line 14 allows an attacker to overwrite the value contained in `user_filename`, which the application assumes was checked by the procedure invoked at line 9. Therefore, the attacker can leverage the overflow to append data of her choice to any file the application has access to. Note that the execution of this data modification attack does not affect the type or ordering of the system calls issued by the application.

To detect data modification attacks, models must include some representation of valid or normal program state. For example, prior work in [11] and [12] uses learning models to characterize "normal" system call argument values and to demonstrate that changes to program state as a result of an attack often manifest themselves as changes to the argument values of system calls. The assumption underlying this approach is that the goal of the attacker is to leverage the privileges of an application to change some security-relevant state in the underlying system (e.g., write chosen values to a file, execute a specific application, or change the permissions of a security-critical file). This type of activity may be readily observed as suspicious system call argument values.

One limitation of the argument modeling approach in [11], [12], and [15] is that models of normal argument values are built for each system call. That is, one set of models is created for `open`, another set for `execve`, and so on. As a result, a model captures the full range of argument values observed during all phases of the execution of an application. A better approach would be to train the models in a way that is specific to individual phases of a program's execution. For example, the arguments used during a program's initialization phase are likely to differ from those used during a production phase or termination phase. This can be achieved by differentiating program behavior using the *calling context* of a procedure – that is, the configuration of the application's call stack when a procedure is invoked. Similar techniques have been explored in the programming languages literature. Examples include improving profiling by considering a procedure's calling context [1], analyzing pointer variables more accurately [9], and improving lifetime predictions of dynamically allocated memory [16]. A common observation in these approaches is that the calling context of a procedure is often a powerful predictor of how the procedure and its data interact.

In this paper, we first propose and evaluate a metric for determining to what extent argument values are unique to a particular call stack for a given application. Our study, presented in Section 2, shows that this is predominantly the case, indicating that the argument modeling approach of [12] can be made more precise if models are built for each calling context in which a system call is issued by an application. Armed with this knowledge, we then introduce and evaluate a model-based detection system that builds separate argument models for each call stack in which an application issues a system call. Our experiments demonstrate that the trained models effectively generalize from the training data, performing well during a subsequent detection period.

This paper makes the following primary contributions:

- It analyzes the relationship between system call arguments and different calling contexts, and it introduces a novel metric to quantify the degree to which argument values exhibit uniqueness across contexts.
- It demonstrates that the application's call stack can be leveraged to add context to the argument values that appear at the system call interface. It also demonstrates that the increased sensitivity of context-specific argument models results in better detection performance.
- It defines a technique to detect data modification attacks, which are not detected by previously proposed approaches based on system call sequences.
- It presents an extensive real-world evaluation encompassing over 44 million system call invocations collected over 64 days from 10 hosts.

The remainder of the paper is structured as follows. In Section 2 we introduce and apply a metric to characterize the degree to which system call argument values are unique to calling contexts in which system calls are issued. Then, in Section 3, we present our detection approach, which builds argument models that are specific to each calling context. Section 4 reports the results of evaluating the system empirically. Section 5 covers related work on system call-based anomaly detection. Finally, Section 6 draws conclusions and outlines future work.

2 System call argument and calling context analysis

The effectiveness of system call analysis that includes call stack information is directly related to the number of contexts in which a given argument value associated with the invocations of a particular system call occurs. More specifically, if argument values appear in many contexts, essentially randomly, context-specific learning models are likely to offer no benefit. Furthermore, if each observed argument value appears (possibly multiple times) in only one context, we would expect system call argument analysis that includes call stack information to outperform context-insensitive models. In this section, we propose a metric to express the degree of context-uniqueness of argument values. We then use this metric to determine which applications are likely to be amenable to system call analysis that takes into account stack-specific behavior.

Before introducing our context-uniqueness metric, we need to define some notation. Let $S = \{s_1, s_2, \ldots\}$ be the set of monitored system calls, and let $A^{s_i} = \langle A_1^{s_i}, \ldots, A_n^{s_i} \rangle$ be the vector of formal arguments for system call s_i. Consistent with [6], we define the calling context of a system call invocation as the sequence of return addresses $C = \langle r_1, \ldots, r_l \rangle$ stored on the application's call stack at the time the system call invocation occurs. Each invocation s_{ij} of s_i has a concrete vector of values for A^{s_i} defined as $a^{s_{ij}} = \langle a_1^{s_{ij}}, \ldots, a_n^{s_{ij}} \rangle$, and two argument vectors $a^{s_{ij}}$ and $a^{s_{ij'}}$ are considered distinct if any of their subvalues $a_l^{s_{ij}}$ and $a_l^{s_{ij'}}$ differ.

We are interested in the set of argument vectors appearing in the invocation of a system call in a particular context. For this, we introduce the notion of an *argument set*. An argument set for a system call s_i in a context C is the set of all argument vectors $a^{s_{ij}}$ observed for the chosen system call when it is issued in the calling context C. This is denoted by $AS(C, s_i)$. The argument set for s_i across the entire application (i.e., ignoring the calling context) is denoted by $AS(*, s_i)$. We observe that if the set $AS(*, s_i)$ is partitioned by the subsets $\{AS(C_1, s_i), AS(C_2, s_i), \ldots\}$, then each recorded argument vector a^{s_i} occurs in only one calling context.

One potential route in the development of this metric would be to adapt cluster quality measures from the machine learning literature. Unfortunately, computing the distance between two argument vectors $a^{s_{ij}}$ and $a^{s_{ij}\prime}$ is problematic. For example, integer arguments that exhibit numeric similarity are often dissimilar in their semantic meaning. This occurs in cases where an integer argument is the logical OR of a collection of boolean flags. Computing string similarity also presents difficulties. For example, two filesystem paths may have large common substrings or a small Hamming distance, but correspond to files that have a very different meaning to the users of the system. For these reasons, we build our metric using argument vector equality only.

With this in mind, we would like to determine the number of contexts where each distinct argument vector is used. To measure this we define the *actual partitioning* value $AP(s_i)$, which is the sum over all recorded concrete argument vectors of the number of argument sets where each $a^{s_{ij}}$ appears during the period of monitoring. That is,

$$AP(s_i) = \sum_{j=1}^{K} \sum_{m=1}^{L} \mid \{a^{s_{ij}}\} \cap AS(C_m, s_i) \mid \tag{1}$$

where K is the number of distinct argument vector values recorded, and L is the number of distinct stack configurations observed during the monitoring period.

For our context-uniqueness metric, we would like to compare the actual partitioning value to both the optimal partitioning and the worst case partitioning values. For the optimal case, each argument vector should appear in as few contexts as possible. There are two cases to consider. In the case where the number of distinct argument vectors is greater or equal to the number of calling contexts ($K \geq L$), each argument value appears in only one context in the optimal partition of $AS(*, s_i)$. For the case when $K < L$, some argument vectors must appear in more than one context[1]. The optimal partitioning, in this case, is for each concrete argument vector to appear in L/K argument sets. Both cases can be expressed by specifying the number of argument sets where each argument vector is to appear as $max(L/K, 1)$.

We can now define the optimal partitioning value and the worst case partitioning value. Since there are K distinct argument vector values, the *optimal partitioning* value $OP(s_i)$ is defined as:

$$OP(s_i) = K * max(L/K, 1) = max(L, K) \tag{2}$$

To define the worst case, we need to know how many instances of each of the K distinct argument vectors $a^{s_{ij}} \in AS(*, s_i)$ were recorded during the monitoring

[1] If each distinct value appeared in only one context, then there would be contexts with no argument vectors.

period. We define the counter $cnt_{a^{s_{ij}}}$ as the number of times that a particular argument vector $a^{s_{ij}}$ occurs in the recorded invocations. The worst case partitioning is determined by distributing each of the K argument vectors in $AS(*, s_i)$ over as many contexts as possible. Although $a^{s_{ij}}$ can appear a maximum of $cnt_{a^{s_{ij}}}$ times, there are only L distinct contexts. Therefore, $a^{s_{ij}}$ appears in $min(cnt_{a^{s_{ij}}}, L)$ argument sets in the worst case partitioning. Thus, the *worst case partitioning* value WP is defined as:

$$WP(s_i) = \sum_{j=1}^{K} min(cnt_{a^{s_{ij}}}, L) \tag{3}$$

Now that we have the actual partitioning value, the optimal partitioning value, and the worst case partitioning value, we can define a measure of the partition quality $Q(s_i)$ for a system call s_i. $Q(s_i)$ is defined as the ratio of the difference between the actual and optimal partitioning to the difference between the worst case and optimal partitioning:

$$Q(s_i) = \frac{AP(s_i) - OP(s_i)}{WP(s_i) - OP(s_i)}$$

Since the actual partitioning $AP(s_i)$ must fall between $WP(s_i)$ and $OP(s_i)$, $Q(s_i)$ takes on values in the interval $[0, 1]$ with 0 being the highest quality partitioning (i.e., no difference from the optimal case) and 1 being the worst (i.e., no difference from the worst case partitioning). In the special case where there is no difference between $WP(s_i)$ and $OP(s_i)$, we define $Q(s_i)$ to be 1.

Context	**Observed argument set**
C_1	$AS(C_1, s_{foo}) = \{$`"/tmp/a", "/tmp/b", "/tmp/c"`$\}$
C_2	$AS(C_2, s_{foo}) = \{$`"/tmp/a"`$\}$
C_3	$AS(C_3, s_{foo}) = \{$`"/tmp/a"`$\}$

Table 1. Observed argument sets for a fictional system call `foo(char *pathname)` in three different calling contexts, C_1, C_2, and C_3.

Consider the example shown in Table 1, which gives observed argument values for a fictional system call
`foo(char *pathname)` for $L = 3$ different calling contexts, C_1, C_2, and C_3. Further, suppose that each argument appears 3 times during the period of monitoring, that is, $cnt_{a^{s_{(foo)j}}} = 3$ for each of the three $s_{(foo)j}$. Since the concrete argument vector $\langle$`"/tmp/a"`$\rangle$ appears in all three contexts and the argument vectors $\langle$`"/tmp/b"`$\rangle$ and $\langle$`"/tmp/c"`$\rangle$ appear in one context each, the actual partitioning $AP(s_{foo})$ is:

$$AP(s_{foo}) = (1 + 1 + 1) + (1 + 0 + 0) + (1 + 0 + 0) = 5 \tag{4}$$

Because $L = K = 3$, the optimal partitioning for s_{foo} is

$$OP(s_{foo}) = max(3, 3) = 3 \tag{5}$$

and the worst case partitioning for s_{foo} is

$$WP(s_{foo}) = min(3, 3) + min(3, 3) + min(3, 3) = 9 \tag{6}$$

Combining the actual, optimal, and worst case partitioning, we have the following measure of the overall quality of the partitioning for s_{foo}:

$$Q(s_{foo}) = \frac{5 - 3}{9 - 3} = 1/3 \tag{7}$$

To evaluate our quality metric, we selected 9 root-owned services and periodic (cron) applications running in a production setting on 10 servers in an undergraduate computer science lab. The 9 audited programs were chosen from a larger pool of processes that run with root privileges in the following way. First, no interactive command line executables were evaluated since they appear sporadically and generate a relatively small number of audit records. For similar reasons, 8 periodic and daemon processes were removed from the study because they did not appear frequently enough in the audit set to produce a meaningful evaluation. Second, script language interpreters (e.g., Perl and Python) were removed since programs implemented in those languages execute with a virtualized call stack. Next, 6 processes associated with the X11 windowing system were eliminated because their role in the system is primarily to facilitate graphical interaction with the user. Finally, 5 programs associated with the package management and compilation subsystem were eliminated because they have a peripheral role with respect to the security of the system.

Table 2 shows the mean and standard deviation of Q values across 36 security-critical system calls issued by each of the 9 programs over a 10-day period. Section 3.1 specifies the monitored system calls and provides further justification for their inclusion in the study. Table 2 tabulates the average (μ) and standard deviation (σ) of Q across each of the 36 system calls (denoted $Q(s_*)$). The data shows that the values of Q recorded for a collection of real applications in a production setting are optimal in 3 of 9 cases, and are never greater than 0.169. This suggests that including call stack information in system call argument analysis is likely to produce models that outperform those that do not consider execution context.

3 System design

The empirical evaluation of context-sensitivity in the previous section showed that system call arguments are often uniquely associated with specific calling

Application	$Q(s_*)\ \mu$	$Q(s_*)\ \sigma$
cfenvd	0.038	0.066
cfexecd	0.107	0.191
crond	0.000	0.000
cupsd	0.085	0.159
idmapd	0.000	0.000
sendmail	0.093	0.194
slocate	0.169	0.379
sshd	0.168	0.218
ypbind	0.000	0.000
Overall	0.063	0.209

Table 2. Mean and standard deviation of Q over all system calls for the nine applications in the study.

context in real-world applications. Therefore, we developed an intrusion detection system that takes advantage of this property. Our approach uses a collection of context-specific learning models that operate in three distinct phases. The first two phases consist of a *training phase* and a *threshold learning phase*, during which learning is performed on attack-free audit data. In the training phase, models gather examples of normal system call arguments. At the end of this phase, detection models are generated for use in the two subsequent phases. Following the training phase is the threshold learning phase, where thresholds are computed for the finalized models by measuring their response to attack-free data. In the final *detection phase*, the trained models and thresholds are used together to classify events as normal or anomalous.

In the following, we describe feature selection and the context-specific modeling approach in Section 3.1. Then, in Sections 3.2 through 3.4, we describe the three phases of system operation. Finally, Section 3.5 provides details about the audit collection infrastructure.

3.1 Feature selection and the context-specific modeling approach

Experience shows that evidence of attacks often appears in the argument values of system calls. Sometimes this may be due to "collateral damage" to local (stack) variables when overwriting a return address. In these cases, damaged variables are then used in system call invocations before the procedure returns. In other cases, the attack is leveraging the privileges of the application to perform actions that are not normally performed by the victim program. In many instances, these differences can be identified by argument models.

To determine the set of system calls to use for our analysis, we studied the 243 system calls implemented in the version 2.6.10 of the Linux kernel to determine which additional calls represent avenues to leveraging or increasing the privilege of applications. This study identified 36 system calls, shown in Table 3, that we found should be monitored to detect attempts to compromise the security of a host. Note that in our system only arguments that have intrinsic semantic meaning are modeled. Integer arguments corresponding to file descriptors and memory addresses, for example, are ignored, since their values are not meaningful across runs of an application. Additionally, these values rarely contain any semantic information about the operation being performed.

open	creat	link	unlink
execve	mknod	chmod	mount
umount	rename	mkdir	rmdir
umount2	symlink	truncate	uselib
ftruncate	fchmod	ioperm	iopl
ipc	mprotect	create_module	prctl
capset	lchown	setreuid	setregid
fchown	setresuid	setresgid	chown
setuid	setgid	setfsuid	setfsgid

Table 3. The 36 system calls monitored by the system.

In order to leverage the context information provided by the application's call stack at the time a system call is invoked, we instantiate detection models for each calling context encountered during the training phase. We rely on audit records that are composed of two parts: (a) the system call s_i that was invoked, along with its arguments $a^{s_{ij}} = \langle a_1^{s_{ij}}, \ldots, a_n^{s_{ij}} \rangle$, and (b) the sequence of return addresses gathered from the application's call stack when the system call was invoked. These addresses form the system call's context $C = \langle r_1, \ldots, r_l \rangle$. In all three phases (training, thresholding, and detection), the pair $\langle C, s_i \rangle$ is used as a lookup key in a data structure that maintains the collection of context-specific models and thresholds.

3.2 Training phase

The first phase of system operation is training, during which the audit records received by the audit daemon are used as examples of normal behavior to train context-specific argument models. This approach improves upon prior work ([12]), which did not consider execution context, but instead applied the same argument model instantiations to all invocations of a particular system call issued by an application.

We now describe the individual argument models used to characterize normal values for system call arguments. The models are described in substantial detail in our previous work; the reader is referred to [12] and [14] for information beyond the brief descriptions provided here.

The following three models are applied to string arguments:

- *String Length:* The goal of the string length model is to approximate the actual (but unknown) distribution of the lengths of string arguments and to detect instances that significantly deviate from the observed normal behavior. Usually, system call string arguments represent canonical file names that point to an entry in the file system. These arguments are commonly used when files are accessed (`open`, `stat`) or executed (`execve`), and their lengths rarely exceed a hundred characters. However, when malicious input is passed to programs, this input often occurs in an argument of a system call with a length of several hundred bytes. The detection of significant deviations is based on the Chebyshev inequality.

– *String Character Distribution:* The string character distribution model captures the concept of a normal string argument by looking at its character distribution. The approach is based on the observation that strings have a regular structure, are often human-readable, and almost always contain only printable characters. In the case of attacks that send binary data, a completely different character distribution can be observed. This is also true for attacks that send many repetitions of a single character (e.g., the `nop`-sledge of a buffer overflow attack). The detection of deviating arguments is performed using a statistical test (Pearson χ^2-test) that determines the probability that the character distribution of a system call argument fits the normal distribution established during the training phase.
– *String Structural Inference:* Often, the manifestation of an exploit is immediately visible in system call arguments as unusually long strings or strings that contain repetitions of non-printable characters. There are situations, however, when an attacker is able to craft her attack in a manner that makes its manifestation appear more regular. For example, non-printable characters can be replaced by groups of printable characters. In such situations, we need a more detailed model of the system call argument. Such a model can be acquired by analyzing the argument's structure. For the purposes of this model, the structure of an argument is the regular grammar that describes all of its normal, legitimate values. The process of inferring the grammar from training data is based on a Markov model and a probabilistic state-merging procedure. The details are presented in [21] and [22].

The fourth model can be used for all types of system call arguments:

– *Token Finder:* The purpose of the token finder model is to determine whether the values of a certain system call argument are drawn from a limited set of possible alternatives (i.e., they are elements or tokens of an enumeration). An application often passes identical values such as flags or handles to certain system call arguments. When an attack changes the normal flow of execution and branches into maliciously injected code, these constraints are often violated. The decision whether to identify the set as an enumeration or a collection of random identifiers can be made utilizing a simple statistical test, such as the non-parametric Kolmogorov-Smirnov variant, as suggested in [13].

In prior work ([12]), models were instantiated for each system call (e.g., `open`, `execve`). As we noted, in this paper models have been replicated for each calling context C. In this way, when the audit daemon is operating in the training phase, aggregate model instances are trained on the observed argument set $AS(C, s_i)$.

3.3 Threshold learning phase

In our design, an *aggregate model* is used to associate a set of models with each system call. The task of an aggregate model is to combine the outputs of all models that are associated with a system call into a single anomaly score that is used to assess whether the entire system call is normal or not. As in [12], we sum the negative logarithm of the individual model outputs to produce one score, which is then compared to a threshold (described below) to determine whether or not an alert should be generated for the system call.

At the start of the threshold-learning phase, training ceases and all models instantiated by the system are switched to detection mode. Each event in the (attack-free) threshold learning set is then assigned an anomaly score by the aggregate model specific to its system call s_i and context C. The threshold for the aggregate model associated with the pair (C, s_i) is computed by adding 20% to the maximum anomaly score generated by the aggregate model over the threshold training set.

Using a context-specific characterization allows thresholds to be independent of one another, permitting some thresholds to be "loose" and others to be "tight". For example, in one context where there are a large number of training examples, models might characterize the context's features virtually flawlessly implying a very tight threshold for that context. In another context that appears far less frequently in the training set, the instantiated models may have a more coarse approximation of the feature values, resulting in a relatively loose threshold.

3.4 Detection phase

When an audit record is received during the detection phase, the system first checks if the context and system call pairing associated with it (that is, $\langle C, s \rangle$) has been observed during the training period. If the pairing was not recorded during the training phase, the system issues an alert. For pairings that were observed during the training phase, the system uses the values for C and s to look up the aggregate model that was created during training, uses the model to evaluate the argument values contained in the audit record, and issues an alert if the resulting score exceeds the threshold associated with $\langle C, s \rangle$.

3.5 Auditing subsystem

This section provides details of the implementation used for the evaluation of the system. The system described in this paper is composed of two modules: a kernel-resident audit module that records system call invocations and the application calling context in which they appear, and a user-space audit daemon that develops models of system call argument values using machine learning techniques. The two components communicate via an entry in the `proc` filesystem.

Both the learning and detection phases require a stream of system call invocation events. System call event auditing is accomplished using an implementation based on the Snare audit module, which is an existing loadable kernel module written for the Linux operating system by Intersect Alliance [20]. This module intercepts system calls through the use of system call interposition, which is realized by overwriting the kernel's table of function pointers to system calls with pointers to wrapper functions. These wrapper functions generate an audit record prior to calling the original system call and before returning its result. To realize the goals of this project, several significant changes were made to the Snare module:

User stack unwinding. When audited system calls are invoked, in addition to recording the arguments to the system call, the user's memory space is probed iteratively to unwind the frames stored on the user application's stack. This process is very similar to the one followed by a debugger as it recovers the stack frames from the memory of a running application.
Virtual addresses encountered on the user's stack are matched against the memory-mapped address ranges maintained in the process control block. When a matching address range is found, the stack address is normalized by subtracting the starting address of the memory-mapped region, and the module records the normalized address along with the i-node of the file containing the memory mapped code. In this way, address consistency is maintained across runs of an application, or in the face of dynamic loading and unloading of code by the application.
Signaling of user audit daemon replaced with support for blocking reads. The original version of the Snare audit module delivered a signal to the audit daemon each time an event was generated. This created performance problems during periods of high load, which were often accompanied by a high volume of audit data. Our version uses a kernel wait queue, which avoids signal storms during periods of heavy load.

4 Empirical validation

Application	Total Events	False Positives	False Positive Rate
cfenvd	11,918,468	0	$0.00 \times 10^{+00}$
cfexecd	457,812	4,407	9.63×10^{-03}
crond	1,265,345	0	$0.00 \times 10^{+00}$
cupsd	291,022	1,942	6.67×10^{-03}
idmapd	57,316	2,962	5.17×10^{-02}
sendmail	5,514,158	1,559	2.97×10^{-04}
slocate	11,914,501	155	1.30×10^{-05}
sshd	13,347,164	1,931	1.45×10^{-04}
ypbind	30,268	0	$0.00 \times 10^{+00}$
Overall	44,796,054	12,956	2.89×10^{-04}

Table 4. False positive rates for models that do not consider calling context.

The purpose of this empirical study is to investigate the impact of considering the calling context of system calls on the detection capability of the system. The evaluation consists of three parts. Section 4.1 compares context-specific models to context-insensitive models with respect to the generation of false positives. Next, Section 4.2 addresses the question of whether context-specific models offer an improvement in precision over context-insensitive models. Section 4.3 evaluates the ability of the system to detect real attacks launched against two monitored applications. Finally, Section 4.4 quantifies the computational overhead of context-sensitive monitoring.

4.1 Comparing context-sensitive and context-insensitive argument models

Since models trained specific to particular calling contexts occurring in an application have a smaller, more restrictive set of training examples, they potentially suffer from the drawback of being too sensitive to variations in argument values observed during the detection phase. Therefore, it is critical to determine their false positive rate relative to context-insensitive argument models.

In order to quantify the rate of false positives observed in practice in each case, we collected audit data on root-owned daemons and periodic (cron) applications running on 10 hosts in an undergraduate computer science instructional laboratory over a period of 64 days. During the recorded period, each of the hosts were accessed regularly by approximately 100 unique users (administrators and undergraduate users) who interacted with the system in local X11 sessions in addition to remote logins. The recorded audit data was checked for known attacks and is, to the authors' knowledge, free of attacks. We also tracked publicly released vulnerabilities on security mailing lists and noted no vulnerabilities in the monitored software. In all cases, the system was trained and evaluated using data collected at each host. In the interest of conciseness, however, detection performance is reported in aggregated form (i.e., measurements are combined from all 10 hosts used in the study).

Of the 64 days of recorded audit data, the first 39 days were used for training the argument models. Thresholds were computed using the following 7 days of audit data, and detection was performed on the final 18 days. The false positives produced by the system for context-insensitive models (i.e., models that ignore calling context) are shown in Table 4, and Table 5 summarizes the false positive rates for context-sensitive models. Separate figures are given for alarms generated for unknown contexts (i.e., contexts that were not seen during the training phase) as well as for alarms generated from anomalous model scores.

From the tabulated data, it is clear that the overall false positive rates of context-sensitive models outperform context-insensitive models by a factor of about 2.7. Further inspection of the 1,007 unknown context alarms for the `cfexecd` application revealed that they were repeated instances of alarms for 40 contexts that did not appear in the training data. Additionally, all 1,122 unknown context alarms issued for the `sendmail` application, and 348 of 379 of the alarms issued for `sshd` each occurred on a single day. This suggests that it would be straightforward for an administrator to add these contexts to the known set and eliminate future instances of those alarms. Taken together, unknown context and model violation alarms represent an average of 34 alarms per application per day. This is a relatively manageable number, and post-processing tools could likely improve this figure by summarizing duplicate alarms [17].

Table 5 shows a large number of model violation alarms (1,705) for the `sshd` application. Further analysis showed that 652 (more than 38%) of those violations were triggered by models for the `setresuid` system call. These anomalous calls were the result of users that had not been observed during the training period logging into the system. In an academic computer network, this level of irregular user behavior can be expected. However, on more sensitive networks, these alarms could be valuable indicators of misuse or misconfiguration of login policies.

Application	Total Events	Unknown Context Alarms	Model Violation Alarms	Overall FP Rate
cfenvd	11,918,468	21	0	1.76×10^{-06}
cfexecd	457,812	1,007	31	2.27×10^{-03}
crond	1,265,345	0	0	$0.00 \times 10^{+00}$
cupsd	291,022	6	252	8.87×10^{-04}
idmapd	57,316	0	0	$0.00 \times 10^{+00}$
sendmail	5,514,158	1,122	154	2.31×10^{-04}
slocate	11,914,501	0	183	1.54×10^{-05}
sshd	13,347,164	379	1,705	1.56×10^{-04}
ypbind	30,268	0	0	$0.00 \times 10^{+00}$
Overall	44,796,054	2,535	2,325	1.09×10^{-04}

Table 5. False positive rates using context-sensitive models.

4.2 Cross-comparison of context-specific models

Section 2 proposed the metric Q for quantifying the degree to which argument values are unique across the various execution contexts in which a particular system call occurs. The experiment described in this section is intended to further validate the context-specific detection approach. The experiment performs cross-comparison of context-sensitive models for system calls s_i on events drawn from all contexts C in which s_i occurs. Whereas Q measured the extent to which the observed argument sets (ASs) are disjoint, this experiment is designed to measure the extent to which learned context-specific models are able to capture these differences.

To show this, the models are trained for each context exactly as described in Section 4.1, but each system call is evaluated not only on the model for its native context, but on all non-native models as well. If context-specific models capture context-specific features, we would expect events to be classified as normal in their native context and as anomalous in all other contexts.

Table 6 shows that context-sensitive models are, in the vast majority of cases, able to correctly classify events as belonging or not belonging to the context for which the model was trained. This evidence supports three conclusions. First, the

Application	Native FP rate	Non-native FP rate
cfenvd	$0.000 \times 10^{+00}$	0.967
cfexecd	6.771×10^{-05}	0.877
crond	$0.000 \times 10^{+00}$	1.000
cupsd	8.660×10^{-04}	0.947
idmapd	$0.000 \times 10^{+00}$	1.000
sendmail	2.793×10^{-05}	0.933
slocate	1.536×10^{-05}	0.974
sshd	1.277×10^{-04}	0.855
ypbind	$0.000 \times 10^{+00}$	1.000
Average	1.105×10^{-04}	0.950

Table 6. Cross-comparison of context-sensitive models. Rate of false positives for events in native and non-native contexts are shown.

calling context of system calls is a strong predictor of the subclass of argument values observed at the system call interface for a number of applications in a real-world, operational setting. Second, learning models are able to capture this differentiated behavior. Finally, the results suggest that context-specific models capture a more restricted range of behavior than context-insensitive models. This implies that context-sensitive models restrict the number of options that an attacker has to influence the arguments of system calls while avoiding detection.

4.3 Measuring the detection capability of call stack-specific argument models

Source code and binary audits were performed for the 9 services and application used in our study, but no vulnerabilities were found. Therefore, in order to measure the attack detection capability of call stack-specific argument models, we tested the system using attacks on a proprietary setuid application as well as on an Apache web server. Following is a description of the attacks and the corresponding detection performance of the system.

Proprietary setuid application An experiment was conducted on a setuid root application installed on the 10 audit hosts used in this study. The program in question is a proprietary setuid root application written to allow students to submit homework assignments to a class account for grading. While this program is not a daemon or periodic job, an analysis of its binary revealed an exploitable stack overflow vulnerability in a request logging function. This vulnerability was used to test the detection capability of our system. The attack on this program required circumventing the exec-shield, stack randomization, and heap randomization protection mechanisms deployed on the monitored hosts. The attack involved overwriting two stack variables: the current function's return address and the frame pointer. This caused the program to jump to an indirect jump instruction through the modified frame pointer, transferring control to an exploit payload previously injected in a buffer on the heap. This was necessary in order to overcome the exec-shield and randomization protection mechanisms. The results and analysis of the context-sensitive detection system's sensitivity to exploit payloads is discussed below.

Rootshell exploit The first exploit payload executed against the vulnerable program was a simple shell execution with root privileges. Because the `execve` system call was invoked from a context not previously observed during the training period, the context-sensitive detection system was able to distinguish the system call invocation as anomalous and report an alert. The detection system configured in context-insensitive mode, however, did not detect the `execve` call as anomalous. This stems from the fact that both a file archiving utility and a compression utility are spawned during the normal execution of the assignment submission program, and thus the context-insensitive argument models on their own were not sensitive enough to detect an anomaly based on the `execve` target alone. A final observation of this scenario is that a sequence-based system call IDS would have detected a deviation from the normal sequence of system calls, and would have raised an alert.

Data modification exploit The second exploit payload executed against the assignment submission program was a variation of a data modification attack. The objective of this exploit was to manipulate the logging of an assignment submission such that the submitter and timestamp could be subverted with attacker-supplied values. To accomplish this, the exploit payload first called `mprotect` from a legitimate, in-sequence context to mark the code segments of the process read/write. Since the stack was modified to hold a legitimate sequence of return addresses prior to calling `mprotect`, the program continued executing native application code upon returning. In order to regain control for the second part of the attack, a system library function pointer was overwritten in the procedure linkage table (PLT). This type of attack is described in detail in [10]. Changing the memory protection bits on the code segment of the program allowed the statically defined format string that is used in the invocation to `fprintf` to be overwritten. In this way, the attacker's format string was used in place of the legitimate one when the transaction was logged by the program.

A sequence-based system call IDS would not have detected an anomaly, as no invalid or out-of-sequence system calls were invoked. In addition, the context-insensitive argument models were not tight enough to detect an aberration in the parameters to the system call `mprotect`. The context-sensitive detection system, however, was able to detect the anomalous argument due to the more precise argument modeling that included system call context.

Detecting attacks against OpenSSL The final demonstration of the attack detection capability of the system involved testing an off-the-shelf exploit for the Apache web server running with a vulnerable version of OpenSSL, which is a popular implementation of the Secure Sockets Layer (SSL) and Transport Layer Security (TLS) protocols. This version of OpenSSL is vulnerable to a remote client master key overflow, allowing an attacker to potentially execute arbitrary code in any network service that utilizes the library.

As before, context-sensitive and context-insensitive model instances were trained against traces of normal HTTP client behavior. The models were then applied to a trace of an attack against OpenSSL. As before, the stack specific models correctly identified the attack, in this case from an anomalous `execve` of "/bin/sh." The context-insensitive models, however, did not consider this system call to be sufficiently anomalous to raise an alarm. We speculate that since the training data included benign invocations of CGI scripts, which necessarily involve issuing an `execve` for an external script execution, the context-insensitive models were not able to differentiate between benign and malicious invocations of the system call. This is because only one profile was constructed from the training set for `execve`, which supports our claim that the detection capability of argument models is measurably enhanced by instantiating models specific to each call stack context.

4.4 Performance overhead of stack unwinding

To evaluate the performance overhead of unwinding the call stacks of user processes, we constructed a benchmark application. The benchmark invokes a system call after creating a parameterized number of frames on the callstack. In each run of the benchmark, 100 groups of 100 such invocations are made and the average time to complete 100 invocations is returned. In Figure 2 we compare the benchmark running times of an identical system in three configurations: no auditing whatsoever, simple system call auditing (no stack unwinding), and system call auditing with stack unwinding. The benchmark execution time is given for a variety of stack depths.

The figure shows that there is significant overhead associated with unwinding user call stacks while auditing. However, the overheads are roughly similar to simple auditing for stack depths less than 40 (i.e., within a factor of two). We also note that the benchmark is designed to expose differences in the audit times, and differs from normal applications in that it does essentially no other processing aside from rapidly invoking system calls.

5 Related work

Research on model-based detection using system call invocation models originated with [4], which analyzes fixed-length sequences of system calls, without considering arguments or return values. The model of legitimate program behavior is built by observing normal system call sequences in attack-free application runs. Alternative data models for the characterization of system call sequences were proposed in [25] and [26].

These detection techniques could be easily evaded by mimicry attacks, in which an exploit is crafted to produce a legitimate sequence of system calls while performing malicious actions [24]. The introduction of gray-box and white-box approaches, which use additional information such as the stack context and information derived through static analysis techniques, have considerably raised

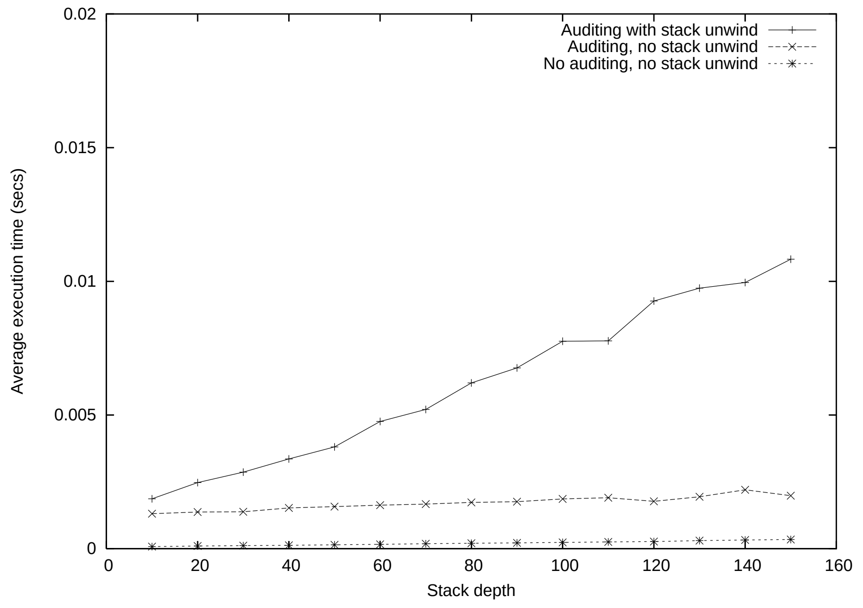

Fig. 2. Comparison of average execution time of system call invocation benchmark.

the bar for this kind of attack [18, 3, 5]. Nevertheless, these approaches do not provide effective modeling of system call arguments, giving the attacker a considerable amount of freedom in crafting an exploit that evades detection. Therefore, black-box, learning-based models that take into account the arguments of system calls were introduced to further limit the ability of an attacker to perform mimicry attacks [12, 15].

The system call automaton proposed in [18] was further extended to include the analysis of system call arguments in [19]. The authors motivate this extension by saying that "clearly, it is not enough to know that something is being written by a program – we need to identify the object being modified by the write [operation]." The difference with respect to our approach is that we perform more sophisticated argument modeling and include the complete function call history instead of only the program counter of the system call. Therefore, we are able to detect data modification attacks as well as deviations from established site-specific behaviors that cannot be statically derived.

A further class of proposals extracts models directly from the program's source code or binary representations using static analysis methods [23, 7, 8, 3, 27]. These systems use static analysis to derive the system call automaton, which is then extended with call stack information to remove impossible paths and increase the precision of the detection process.

6 Conclusions and future work

In this paper, we presented a novel approach to the detection of anomalous system calls. Different from previous approaches, our solution combines dynamic stack context analysis with the characterization of system call arguments. The resulting context-sensitive system call model is effective against data modification attacks, which do not modify the sequence of system calls executed by vulnerable applications. It also improves upon the false positive rates of models that only operate on argument values and ignore context information.

We have also introduced a metric that quantifies the degree to which system call arguments are unique to particular execution contexts. Applying this metric to a number of programs deployed in a production setting showed that the set of argument values is optimally or nearly optimally partitioned by the argument sets associated with individual stack configurations. Future work will explore the utility of applying this metric to other intrusion detection domains.

The use of system call argument modeling is orthogonal with respect to analysis techniques that characterize system call sequences. In future work, we will explore how the two approaches can be composed to achieve even more precise detection and better resilience to mimicry attacks[2].

References

1. G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'97)*, 1997.
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 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 IEEE Symposium on Security and Privacy*, May 2003.
4. S. Forrest. A Sense of Self for UNIX Processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 120–128, Oakland, CA, May 1996.
5. D. Gao, M. Reiter, and D. Song. Gray-Box Extraction of Execution Graphs for Anomaly Detection. In *Proceedings of ACM CCS*, pages 318–329, Washington, DC, USA, October 2004.
6. D. Gao, M. Reiter, and D. Song. On Gray-Box Program Tracking for Anomaly Detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 103–118, San Diego, CA, USA, August 2004.
7. J. Giffin, S. Jha, and B. Miller. Detecting Manipulated Remote Call Streams. In *Proceedings of the 11th USENIX Security Symposium*, pages 61–79, 2002.
8. J. Giffin, S. Jha, and B.P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the 11th Network and Distributed System Security Symposium*, San Diego, California, February 2004.
9. M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages*, 21(4), July 1999.

[2] This research was supported by the Army Research Office, under agreement DAAD19-01-1-0484, and by the National Science Foundation, under grants CCR-0238492 and CCR-0524853.

10. C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium*, July 2005.
11. C. Kruegel, D. Mutz, W.K. Robertson, and F. Valeur. Bayesian Event Classification for Intrusion Detection. In *Proceedings of ACSAC 2003*, Las Vegas, NV, December 2003.
12. C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the Detection of Anomalous System Call Arguments. In *Proceedings of the* 8^{th} *European Symposium on Research in Computer Security (ESORICS '03)*, LNCS, pages 326–343, Gjovik, Norway, October 2003. Springer-Verlag.
13. S. Lee, W. Low, and P. Wong. Learning Fingerprints for a Database Intrusion Detection System. In *Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS)*, 2002.
14. D. Mutz. *Context-sensitive Multi-model Anomaly Detection.* Ph.d. thesis, UCSB, June 2006.
15. D. Mutz, F. Valeur, C. Kruegel, and G. Vigna. Anomalous System Call Detection. *ACM Transactions on Information and System Security*, 9(1):61–93, February 2006.
16. E. Nystrom, H. Kim, and W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of Program Analysis for Software Tools and Engineering*, 2004.
17. W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer. Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks. In *Proceeding of NDSS*, San Diego, CA, February 2006.
18. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2001.
19. R. Sekar, V. Venkatakrishnan, S. Basu, Bhatkar S, and D. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003.
20. SNARE - System iNtrusion Analysis and Reporting Environment. `http://www.intersectalliance.com/projects/Snare`.
21. A. Stolcke and S. Omohundro. Hidden Markov Model Induction by Bayesian Model Merging. *Advances in Neural Information Processing Systems*, 1993.
22. A. Stolcke and S. Omohundro. Inducing probabilistic grammars by bayesian model merging. In *Proceedings of the International Conference on Grammatical Inference*, 1994.
23. D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2001. IEEE Press.
24. D. Wagner and P. Soto. Mimicry Attacks on Host-Based Intrusion Detection Systems. In *Proceedings of ACM CCS*, Washington DC, USA, November 2002.
25. C. Warrender, S. Forrest, and B.A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 133–145, 1999.
26. A. Wespi, M. Dacier, and H. Debar. Intrusion Detection Using Variable-Length Audit Trail Patterns. In *Proceedings of RAID*, Toulouse, France, 2000.
27. H. Xu, W. Du, and S. Chapin. Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, Sophia Antipolis, France, 2004.