Behavioral Distance for Intrusion Detection

Debin Gao[1], Michael K. Reiter[2], and Dawn Song[2]

[1] Electrical & Computer Engineering Department, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
`dgao@ece.cmu.edu`
[2] Electrical & Computer Engineering Department, Computer Science Department, and CyLab, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
`{reiter,dawnsong}@cmu.edu`

Abstract. We introduce a notion, *behavioral distance*, for evaluating the extent to which processes—potentially running different programs and executing on different platforms—behave similarly in response to a common input. We explore behavioral distance as a means to detect an attack on one process that causes its behavior to deviate from that of another. We propose a measure of behavioral distance and a realization of this measure using the system calls emitted by processes. Through an empirical evaluation of this measure using three web servers on two different platforms (Linux and Windows), we demonstrate that this approach holds promise for better intrusion detection with moderate overhead.
Keywords: intrusion detection, system call, behavioral distance

1 Introduction

Numerous attacks on software systems result in a process' execution deviating from its normal behavior. Prominent examples include code injection attacks on server processes, resulting from buffer overflow and format string vulnerabilities. A significant amount of research has sought to detect such attacks through monitoring the behavior of the process and comparing that behavior to a model of "normal" behavior. Typically this model of "normal" is obtained either from the process' own previous behavior [10, 27, 9, 8, 13, 12, 37] or the behavior prescribed by the source code or executable of the program it executes [35, 14, 15].

In this paper we present a new approach for detecting anomalous behavior of a process, in which the model of "normal" is a "replica" of the process running in parallel with it, operating on the same inputs. At a high level, our goal is to detect any behavioral deviation between replicas operating on the same inputs, which will then indicate that one of the replicas has been compromised. As we will show, this approach will better detect mimicry attacks [36, 31] than previous approaches. In addition, this approach has immediate application in fault-tolerant systems, which often run replicas and compare their responses (not behavior) to client requests to detect (e.g., [29, 3, 2]) or mask (e.g., [17, 25, 4, 39]) faults. When considering attacks, it is insufficient to simply compare the responses to detect faults, because certain intrusions may not result in observable deviation in the responses (but may nevertheless go on to attack the interior network, for example). Our method of detecting behavioral deviation between replicas can significantly improve the resilience of such fault-tolerant systems by detecting more stealthy attacks.

Monitoring for deviations between replicas would be a relatively simple task if the replicas were identical. However, in light of the primary source of attacks with which we are concerned—i.e., software faults and, in particular, code injection attacks that would corrupt identical replicas identically—it is necessary that the "replicas" be as diverse as possible. We thus take as our goal the task of measuring the behavioral distance between two diverse processes, be they distinct implementations of the program (e.g., as in n-version programming [5]), the same implementation running on different platforms (e.g., one Linux, one Windows), or even distinct implementations on diverse platforms. In this paper, we propose a method to measure behavioral distance between replicas and show that our method can work with competing, off-the-shelf, diverse implementations without modification.

We can measure behavioral distance using many different observable attributes of the replicas. As a concrete example, the measure of "behavior" for a replica that we adopt is the sequence of system calls it emits, since a process presumably is able to affect its surroundings primarily through system calls. Because the replicas are intentionally diverse, even how to define the "distance" between the system call sequences they induce is unclear. When the replicas execute on diverse platforms, the system calls supported are different and not in one-to-one correspondence. When coupled with distinct implementations there is little reason to expect any similarity whatsoever between the system call sequences induced on the platforms when each processes the same request.

A key observation in our work, however, is that even though the system call sequences might not be similar in any syntactic way, they will typically be correlated in the sense that a particular system call subsequence emitted by one replica will usually coincide with a (but syntactically very different) subsequence emitted by the other replica. These correlations could be determined either through static analysis of the replica executables (and the libraries), or by first subjecting the replicas to a battery of well-formed (benign) inputs and observing the system call sequences induced coincidentally. The former is potentially more thorough, but the latter is more widely applicable, being unaffected by difficulties in static analysis of binaries for certain platforms[1] or, in the future,

[1] For example, the complexity of static analysis on x86 binaries is well documented. This complexity stems from difficulties in code discovery and module discovery [24], with numerous contributing factors, including: variable instruction size (Prasad and Chiueh claim that this renders the problem of distinguishing code from data undecidable [22]); hand-coded assembly routines, e.g., due to statically linked libraries, that may not follow familiar source-level conventions (e.g., that a function has a single entry point) or use recognizable compiler idioms [26]; and indirect branch instructions such as `call/jmp reg32` that make it difficult or impossible to identify the target location [24, 22]. Due to these issues and others, x86 binary analysis tools have strict restrictions on their applicable targets [24, 18, 26, 22].

of software obfuscated to render static analysis very difficult for the purposes of digital rights management (e.g., [7]). So, we employ the latter method here.

1.1 Comparison with related work

Utilizing an intrusion detection system to monitor the system calls of a single (non-replicated) process is a thoroughly explored alternative to the approach we explore here for detecting software faults and code-injection attacks. However, all such techniques of which we are aware are vulnerable to *mimicry attacks*, whereby code injected by an attacker issues attack system calls within a longer sequence that is consistent with normal behavior of the program [36, 31, 13]. In the same fashion, independent system call monitoring of each of two diverse replicas does not address this problem, provided that the code injected successfully into one replica uses mimicry. However, as we will show, the alternative we consider here, in which replicas are monitored in a coordinated fashion, makes such an attack far more difficult. The reason is that mimicry of any valid system call sequence on a replica is not sufficient to avoid detection. Rather, to remain undetected, mimicry must induce a system call sequence that is typically observed coincidentally with the sequence emitted by the other, uncorrupted replica.

Viewed more broadly, our approach can be considered a form of intrusion detection that seeks to correlate events from multiple distinct components of a system. Often these events are themselves intrusion detection alerts (e.g., [33, 21]); in contrast, in our approach the events are system calls produced in the course of the system running normally. As such, our work bears a conceptual similarity to other efforts that correlate seemingly benign events across multiple systems to identify intrusions (e.g., [30, 6, 38]). However, we are unaware of any that demonstrate this capability at the system call level.

1.2 Contributions

In this paper we introduce the notion of behavioral distance for intrusion detection, and detail the design, implementation and performance of a system for dynamically monitoring the behavioral distance of diverse replicas. We detail our measure of behavioral distance and our method for divining the correlated system call subsequences of two replicas. We show through empirical analysis with three different `http` server implementations and two different platforms (Linux and Windows) that thresholds for behavioral distance can typically be set so as to induce low false positive (i.e., false alarm) rates while detecting even a minimal attack consisting of merely an `open` and a `write`—even if the attacker knows that our defense is being used. Moreover, the false alarm rate can be further reduced in exchange for some possibility of an attack going undetected (a false negative), though we believe that this tradeoff can be tuned to detect the richer attacks seen in practice with virtually no false alarms. Perhaps more importantly, as a first step in analyzing the behavioral distance of diverse implementations and platforms, we believe this work can lay the framework for future research to improve this tradeoff further.

2 The Problem

The *behavioral distance* that we define should detect semantic similarity/difference when replicas process the same input. That is, provided that replicas process responses in the same way semantically, the behavioral distance should be small. However, because the two replicas may be constructed differently and may run on different operating systems, the two execution traces will naturally differ syntactically. To bridge this apparent discrepancy, we use the fact that since the replicas process the same input, during normal program execution the two syntactically-different executions should represent the same semantic action.

So, our problem is as follows: let s_1 and s_2 denote sequences of observed behaviors of two replicas, respectively. We need to define (and train) a distance measure $\mathsf{Dist}(s_1, s_2)$ that returns a small value when the replicas operate semantically similarly, and returns a large value when they semantically diverge. The quality of the distance measure function $\mathsf{Dist}()$ directly impacts the false positive and false negative rates of the system.

To the best of our knowledge, the problem of developing an accurate behavioral distance measure for detecting software faults and intrusions has not been studied before. Some techniques have been developed to evaluate the semantic equivalence of two sequences of program instructions, though these techniques are difficult to scale to large programs. Also, the problem of semantic equivalence is different from the behavioral distance problem that we study here, since diverse replicas may not behave in exactly the same way. We thus believe we are the first to pose and explore this problem. We also believe that research on this topic could lead to other applications.

There are many ways to monitor the "behavior" of a process. For example, one could look at sequence of instructions executed, or patterns in which process's internal states change. In this paper, we propose a specific measure for behavioral distance, by using system call sequences emitted by processes. A system call is a service provided by the operating system and has been used in many intrusion/anomaly detection systems [10, 27, 9, 8, 13, 12, 37, 35, 14, 15]. It is a reliable way of monitoring program behavior because in most modern operating systems, a system call is the only way for user programs to interact with the operating system. Also, system calls are natural places to intercept a program and perform monitoring, since system calls often require a context switch. Thus, system call monitoring could introduce lower overhead than intercepting the program at other points for monitoring.

3 Behavioral Distance Using System Call Sequences

In this section, we describe how we construct the behavioral distance measure using system call sequences. The goal is to design a quantitative measure such that system call sequences resulting from the same/similar behavior on replicas will have a small "distance" value, and system call sequences resulting from different behavior will have a large "distance" value. As pointed out in Section 1, our objective is to develop such a distance measure *without* analyzing the program source code or executable, i.e., the distance measure function $\mathsf{Dist}(s_1, s_2)$ is defined by first subjecting the server replicas to a battery of well-formed (benign) requests and observing the system call sequences induced.

3.1 Overview

Defining such a behavioral distance measure based on system call sequences is non-trivial. A system call observed is simply an integer, which is the system call ID used in the operating system and carries little meaning.[2] The two replicas may run on two different operating systems such as Linux and Windows; therefore the same system call ID is likely to mean very different things on two different operating systems. However, because the replicas process the same request and generate the same response, there is a strong correlation on the semantics of the system call sequences made by the replicas. Thus, we can evaluate the behavioral distance by identifying the semantic correspondence of the syntactically unrelated system call sequences.

The sequence of system calls made by a replica can be broken into subsequences of system calls, which we call *system call phrases.* A system call phrase is a subsequence of system calls that frequently appear together in program executions, and thus might correspond to a specific task on the operating system or a basic block in the program's source code. If we can learn the correspondence between these phrases, i.e., phrases on two replicas that perform the same/similar task, we can then break sequences of system calls into phrases, and compare the corresponding phrases to find the behavioral distance. A large behavioral distance indicates an attack or a fault on one of the replicas.

Motivated by the above intuition, we propose to calculate the behavioral distance as follows. We first obtain a *distance table*, which indicates the *distance* between any two system call phrases from two replicas. Ideally, the distance associated with two phrases that perform the same task is low, and otherwise is high. Next, we break system call sequences s_1 and s_2 into sequences of system call phrases. (Details are covered in Section 3.5.) The two sequences may have different numbers of phrases, and the corresponding phrases (those that perform similar tasks) might not be at the same location in the two sequences. We handle this problem by inserting *insertion/deletion phrases* (denoted as I/D phrases or σ in the following sections) to obtain two equal-length sequences of phrases $\langle s_{1,1}, \ldots, s_{1,n} \rangle$ and $\langle s_{2,1}, \ldots, s_{2,n} \rangle$. We then look up the distances between the corresponding phrases in the distance table and compute the behavioral distance as the sum of these distances: $\sum_{1 \le i \le n} \mathsf{dist}(s_{1,i}, s_{2,i})$.

In the rest of this section, we first explain more formally how we calculate the behavioral distance, and then describe how we obtain the distance table through learning. Finally we briefly explain how we identify the system call phrases by pattern extraction.

[2] We could consider the arguments to system calls as well, which would supply additional information (e.g., [16]). However, we leave this to future work.

3.2 Behavioral Distance Calculation

In this subsection, we first give the intuition behind our approach by explaining a related problem in molecular biology and evolution. We then formally define our behavioral distance calculation.

A related problem to behavioral distance has been studied in molecular biology and evolution. Roughly speaking, the problem is to evaluate evolutionary change between DNA sequences. When two DNA sequences are derived from a common ancestral sequence, the descendant sequences gradually diverge by changes in the nucleotides. For example, a nucleotide in a DNA sequence may be substituted by another nucleotide over time; a nucleotide may also be deleted or a new nucleotide can be inserted.

To evaluate the evolutionary change between DNA sequences, Sellers [28] proposed a distance measure called *evolutionary distance*, by counting the number of nucleotide changes (including substitutions, deletions and insertions) and summing up the corresponding distances of substitutions, deletions and insertions. The calculation is easy when nucleotides in the two sequences are aligned properly, i.e., corresponding nucleotides are at the same location in the two sequences. However, it becomes complicated when there are deletions and/or insertions, because the nucleotides are misaligned. Therefore, the correct alignment needs to be found by inferring the locations of deletions and insertions. Figure 1 shows an example with two nucleotide sequences and a possible alignment scheme [20].

Original Sequence	Aligned Sequence
`ATGCGTCGTT`	`ATGC-GTCGTT`
`ATCCGCGAT`	`AT-CCG-CGAT`

Fig. 1. Example of two nucleotide sequences

Our behavioral distance calculation is inspired by the evolutionary distance method proposed by Sellers [28], where the evolutionary distance is calculated as the sum of the costs of substitutions, deletions and insertions. In behavioral distance calculations, we also have the "misalignment" problem. Misalignments between system call phrases are mainly due to the diverse implementations or platforms of the replicas. For example, the same task can be performed by different numbers of system call phrases on different replicas. Figure 2 shows an example with two sequences of system call phrases observed when two replicas are processing the same request. Due to implementation differences, s_2 has an extra system call phrase $idle_2$ which does not perform any critical operation.

To calculate the behavioral distance, we thus need to perform an *alignment* procedure by inserting I/D phrases (inserting an I/D phrase in one sequence is equivalent to deleting a corresponding phrase from the other sequence) so that

$$s_1 = \langle open_1, read_1, write_1, close_1 \rangle$$
$$s_2 = \langle open_2, read_2, idle_2, write_2, close_2 \rangle$$

Fig. 2. Example of system call sequences observed on two replicas

system call phrases that perform similar tasks will be at the same position in the two aligned sequences. Given a "proper" alignment, we can then calculate the sum of the distances between the phrases at the same position (Section 3.3 discusses how we obtain the distances between any two phrases) in the two sequences and use this sum as the behavioral distance.

Given a pair of misaligned system call sequences, there are obviously more than one way of inserting I/D phrases into the sequences. Different ways of inserting them will result in different alignments and hence different behavioral distances between the two sequences. What we are most interested in here is to find the behavioral distance between two sequences when the phrases are aligned "properly", i.e., when phrases that perform similar tasks are aligned to each other. Although it is not clear how to find such an alignment for any given pair of sequences, we know that the "best" alignment should result in the smallest behavioral distance between the two sequences, among all other ways of inserting I/D phrases, because phrases that perform similar tasks have a low behavioral distance, as explained in Section 3.3. Therefore, we consider different alignments and choose the one that results in the smallest as the behavioral distance between the two sequences.

Assume that a sequence of system calls s is given in the form of a sequence of system call phrases. Let $\mathsf{prs}(s)$ denote the number of system call phrases in the sequence. Given two sequences s_1 and s_2, we define $\mathsf{Ext}(s_i, n)$ as the set of sequences obtained by inserting $n - \mathsf{prs}(s_i)$ I/D phrases into s_i, at any locations ($i \in \{1, 2\}$). $n = f_1(\mathsf{prs}(s_1), \mathsf{prs}(s_2))$ is the length of the extended sequences after inserting I/D phrases. In order to give more flexibility in the phrase alignments, $f_1()$ ensures that $n > \max(\mathsf{prs}(s_1), \mathsf{prs}(s_2))$. (The definition of $f_1()$ used in our experiments is shown in Section 3.6.)

We define the *behavioral distance* between two system call sequences s_1 and s_2 as

$$\mathsf{Dist}(s_1, s_2) = \min_{s'_1, s'_2} \sum_{i=1}^{n} \mathsf{dist}(s'_{1,i}, s'_{2,i})$$

where

$$s'_1 \in \mathsf{Ext}(s_1, n)$$
$$s'_2 \in \mathsf{Ext}(s_2, n)$$
$$s'_{1,i} \text{ is the } i^{th} \text{ phrase in } s'_1$$
$$s'_{2,i} \text{ is the } i^{th} \text{ phrase in } s'_2.$$

The minimum is taken over all possible values of s'_1 and s'_2. $\mathsf{dist}()$ is the entry in the distance table, which defines the distance between any two phrases from the two replicas. (Section 3.3 discusses how we obtain the distance table. Here we assume that the distance table is given.)

For example, in the case where each phrase is of length one, the calculation of $\mathsf{Dist}(s_1, s_2)$ from the example in Figure 2 may indicate that the minimum is obtained when

$$s'_1 = \langle open_1, read_1, \sigma, write_1, close_1 \rangle$$
$$s'_2 = \langle open_2, read_2, idle_2, write_2, close_2 \rangle.$$

3.3 Learning the Distance Table

The calculation of behavioral distance shown in Section 3.2 assumes that the distances between any two system call phrases are known. In this subsection, we detail how we obtain the distance table by learning. To make the explanations clearer, we assume that the two replicas are running Linux and Microsoft Windows[3] operating systems.

One way to obtain the distance table is to analyze the semantics of each phrase and then manually assign the distances according to the similarity of the semantics. There are several difficulties with this approach. First, this is labor intensive. (Note that the set of system call phrases is likely to be different for different programs.) Second, the information may not be available, e.g., most system calls are not documented in Windows. Third, even if they are well documented, e.g., as in Linux, the distances obtained in this way will be general to the operating system, and might not be able to capture any specific features of the program running on the replicas.

Instead, we propose an automatic way for deriving the distance table by learning. As pointed out in Section 1, our objective is to find the correlation between system call phrases by first subjecting the server replicas to a battery of well-formed (benign) requests and observing the system calls induced. We use the pairs of system call sequences (i.e., system call sequences made by the two replicas when processing the same request) in the training data to obtain the distance table, which contains distances between any two system call phrases observed in the training data. To do that, we first initialize the distance table, and then run a number of iterations to update the entries in the distance table. The iterative process stops when the distance table converges, i.e., when the distance values in the table change by only a small amount for a few consecutive iterations. In each iteration, we calculate the behavioral distance between any system call sequence pairs in the training data (using the modified distance values from the previous iteration), and then use the results of the behavioral distance calculation to update the distance table. We explain how we initialize and update the distance table in the following two subsections.

[3] System calls in Microsoft Windows are usually called native API or system services. In this paper, however, we use the term "system call" for both Linux and Microsoft Windows for simplicity.

Initializing the Distance Table The initial distance values in the distance table play an important role in the performance of the system. Improper values might result in converging to a local minimum, or slower convergence. We introduce two approaches to initialize these distances. We use the first approach to initialize entries in the distance table that involve system calls for which we know the behavior, and use the second approach for the rest. Intuitively, distance between phrases that perform similar tasks should be assigned a small value.

The first approach The first approach to initialize these distances is by analyzing the semantics of individual system calls in Linux and Windows. We first assign similarity values to each pair of system calls in Linux and Windows. Let C^L and C^W be the set of system calls in Linux and Windows, respectively. We analyze each Linux system call and Windows system call and assign a value to $\mathsf{sim}(c_i^L, c_j^W)$, where $c_i^L \in C^L$ for all $i \in \{1, 2, \ldots, |C^L|\}$ and $c_j^W \in C^W$ for all $j \in \{1, 2, \ldots, |C^W|\}$. System calls that perform similar functions are assigned a small similarity value. We then initialize the distances between two system call phrases based on these similarity values.

Let P^L and P^W be the set of Linux system call phrases and Windows system call phrases observed, respectively. We would like to calculate $\mathsf{dist}(p_i^L, p_j^W)$, i.e., the distance between two phrases where $p_i^L \in P^L$ and $p_j^W \in P^W$. (Let $\mathsf{dist}_0(p_i^L, p_j^W)$ denote the initial distance.) We use $\mathsf{len}(p)$ to denote the number of system calls in a phrase p. $\mathsf{dist}_0(p_i^L, p_j^W)$ can now be calculated as

$$\begin{aligned}&\mathsf{dist}_0(p_i^L, p_j^W)\\ =&f_2\left(\{\mathsf{sim}(p_{i,k}^L, p_{j,l}^W) \mid k \in \{1, 2, \ldots, \mathsf{len}(p_i^L)\}; l \in \{1, 2, \ldots, \mathsf{len}(p_j^W)\}\}\right)\end{aligned}$$

where

$$\begin{aligned}&p_{i,k}^L \in C^L && \text{is the } k^{th} \text{ system call in phrase } p_i^L\\ &p_{j,l}^W \in C^W && \text{is the } l^{th} \text{ system call in phrase } p_j^W.\end{aligned}$$

Intuitively, if system calls in the two phrases have small similarity values with each other, the distance between the two phrases should be low. (The definition of $f_2()$ used in our experiments is shown in Section 3.6.)

The main difficulty of this approach is that Windows system calls are not well documented. We have managed to obtain the system call IDs of 94 exported Windows system calls with their function prototypes [19].[4] We then assign distances to these 94 Windows system calls and the Linux system calls by comparing their semantics. Since we do not know the system call IDs and semantics of the rest of the Windows system calls, we propose a second method to initialize the distance table for phrases that involve the rest of the system calls.

[4] Nebbett [19] lists 95 exported Windows system calls, but we only managed to find 94, which are not exactly the same as those listed by Nebbett.

The second approach The second approach to initialize the distance between two phrases is to use frequency information. Intuitively, if two system call phrases perform similar tasks on two replicas, they will occur in the system call sequences in the training data with similar frequencies. We obtain the frequency information when the phrases are first identified by a phrase extraction algorithm and a phrase reduction algorithm; see Section 3.5. The phrase extraction algorithm analyzes system call sequences from sample executions, and outputs a set of system call phrases. The phrase reduction algorithm takes this result and outputs a subset of the system call phrases that are necessary to "cover" the training data, in the sense described below.

The phrase reduction algorithm runs a number of rounds to find the minimal subset of system call phrases identified by the phrase extraction algorithm that can cover the training data. Each round in the phrase reduction algorithm outputs one system call phrase that has the highest coverage (number of occurrences times length of the phrase) in the training data. After the phrase with the highest coverage is found in each round, the system call sequences in the training data are modified by removing all occurrences of that phrase. The phrase reduction algorithm terminates when the training data becomes empty. Let $\mathsf{cnt}(p_i^L)$ and $\mathsf{cnt}(p_j^W)$ denote the number of occurrences of phrases p_i^L and p_j^W in the training data when they are identified and removed by the phrase reduction algorithm, and let $\mathsf{cnt}(P^L)$ and $\mathsf{cnt}(P^W)$ denote the total number of occurrences of all phrases. The frequency with which phrases p_i^L and p_j^W are identified can be calculated as $\frac{\mathsf{cnt}(p_i^L)}{\mathsf{cnt}(P^L)}$ and $\frac{\mathsf{cnt}(p_j^W)}{\mathsf{cnt}(P^W)}$, respectively.

The idea is that system call phrases identified with similar frequencies in the training data are likely to perform the same task, and therefore will be assigned a lower distance.

$$\mathsf{dist}_0(p_i^L, p_j^W) = f_3\left(\frac{\mathsf{cnt}(p_i^L)}{\mathsf{cnt}(P^L)}, \frac{\mathsf{cnt}(p_j^W)}{\mathsf{cnt}(P^W)}\right).$$

$f_3()$ compares the frequencies with which phrases p_i^L and p_j^W are identified and assigns a distance accordingly. (The definition of $f_3()$ that we use in our experiments is shown in Section 3.6.) Distances between a system call phrase and the I/D phrase σ are assigned a constant. $\mathsf{dist}(\sigma, \sigma)$ is always zero.

Iteratively Updating the Distance Table In this subsection, we show how we use the system call sequences in the training data to update the distance table iteratively. We run a number of iterations. The distances are updated in each iteration, and the process stops when the distance table converges, i.e., when the distance values in the table change by only a small amount in a few consecutive iterations. In each iteration, we first calculate the behavioral distance between any pairs of system call sequences (i.e., system call sequences made by the two replicas when processing the same request) in the training data, using the updated distance values from the previous iteration, and then use the results of the behavioral distance calculation to update the distance table.

Note that the result of the behavioral distance calculation not only gives the minimum of the sum of distances over different alignment schemes, but also the particular alignment that results in the minimum. Thus, we analyze the result of the behavioral distance calculation to find out the frequencies with which two phrases are aligned to each other, and use this frequency information to update the corresponding value in the distance table.

Let $\mathsf{occ}_z(p_i^L, p_j^W)$ denote the total number of times that p_i^L and p_j^W are aligned to each other in the results of the behavioral distance calculation in the z^{th} iteration. We then update $\mathsf{dist}(p_i^L, p_j^W)$ as

$$\mathsf{dist}_{z+1}(p_i^L, p_j^W) = f_4\left(\mathsf{dist}_z(p_i^L, p_j^W), \mathsf{occ}_z(p_i^L, p_j^W)\right).$$

Intuitively, the larger $\mathsf{occ}_z(p_i^L, p_j^W)$ is, the smaller $\mathsf{dist}_{z+1}(p_i^L, p_j^W)$ should be. (The definition of $f_4()$ used in our experiments is shown in Section 3.6.) $\mathsf{dist}(p_i^L, \sigma)$ and $\mathsf{dist}(\sigma, p_j^W)$ are updated in the same way, and $\mathsf{dist}(\sigma, \sigma) = 0$.

After the distances are updated, we start the next iteration, where we calculate the behavioral distances between system call sequences in the training data using the new distance values. The process of behavioral distance calculation and distance table updating repeats until the distance table converges, i.e., when the distance values in the table change by a small amount for a few consecutive iterations.

3.4 Real-time Monitoring

After obtaining the distance table by learning, we use the system for real-time monitoring. Each request from a client is sent to both replicas, and such a request results in a sequence of system calls made by each replica. We collect the two system call sequences from both replicas in real time and calculate the behavioral distance between the two sequences. If the behavioral distance is higher than a threshold, an alarm is raised.

3.5 System Call Phrases

Before we start calculating the behavioral distance, we need to break a system call sequence into system call phrases. System call phrases have been used in intrusion/anomaly detection systems [37, 13]. Working on system call phrases significantly improves the performance of behavioral distance calculation, since a relatively long system call sequence is recognized as a short sequence of system call phrases.

We use the phrase extraction algorithm TEIRESIAS [23] and the phrase reduction algorithm in [37], which are also used in intrusion/anomaly detection systems [37, 13], to extract system call phrases. The TEIRESIAS algorithm analyzes system call sequences from sample executions, and outputs a set of system call phrases that are guaranteed to be maximal [23]. Maximal phrases (the number of occurrences of which will decrease if the phrases are extended to include any additional system call) capture system calls that are made in a fixed sequence, and therefore intuitively should conform to basic blocks/functions in the program source code. The phrase reduction algorithm takes the result from TEIRESIAS and outputs a subset of the system call phrases that are necessary to cover the training data. Note that other phrase extraction and reduction algorithms can be used.

For any given system call sequence, there might be more than one way of breaking it into system call phrases. Here we consider all possible ways of breaking it for the behavioral distance calculation and use the minimum as the result. We also group repeating phrases in a sequence and consider only one occurrence of such phrase. The objective is not to "penalize" requests that require longer processing. For example, `http` requests for large files normally result in long system call sequences with many repeating phrases.

3.6 Parameter Settings

The settings of many functions and parameters may affect the performance of our system. In particular, the most important ones are the four functions $f_1()$, $f_2()$, $f_3()$ and $f_4()$. There are many ways to define these functions. Good definitions can improve the performance, especially in terms of the false positive and false negative rates. Below we show how these functions are defined in our experiments. We consider as future work to investigate other ways to define these functions, in order to improve the false positive and false negative rates.

These functions are defined as follows in our experiments:

$$\begin{aligned} f_1(x,y) &= \max(x,y) + 0.2\min(x,y) \\ f_2(X) &= m \ \mathrm{avg}(X) \\ f_3(x,y) &= m(|x-y|) \\ f_4(x,y) &= m(0.8x + 0.2m'y) \end{aligned}$$

where m and m' are normalizing factors used to keep the sum of the costs in the distance table constant in each iteration.

4 Evaluations and Discussions

In this section we evaluate an implementation of our system. We show that the system is able to detect sophisticated mimicry attacks with a low false positive rate. We also show that the performance overhead of our system is moderate.

4.1 Experimental Setup

We setup a system with two replicas running two webservers and one proxy to serve `http` requests. Replica **L** runs Debian Linux on a desktop computer with a 2.2 GHz Pentium IV processor, and replica **W** runs Windows XP on a desktop computer with a 2.0 GHz Pentium IV processor. We use another desktop computer with a 2.0 GHz Pentium IV processor to host a proxy server **P**. All the three machines have 512 MB of memory. The Linux kernel on **L** is modified such that system calls made by the webserver are captured and sent to **P**. On **W**, we develop a kernel driver to capture the system calls made by the webserver. A user program obtains the system calls from the kernel driver on **W** and sends them to **P**.

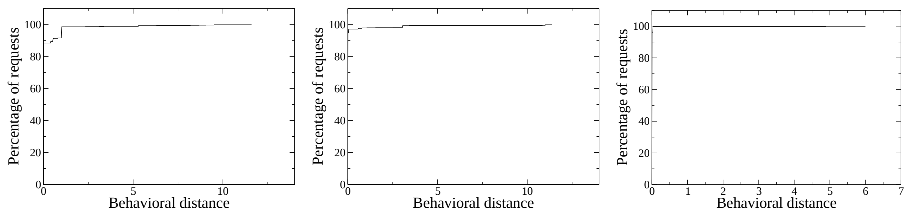
P accepts client `http` requests and forwards them to both **L** and **W**. After processing the requests, **L** and **W** send out responses and the system call sequences made by the server programs. **P** calculates the behavioral distance between the two system call sequences, raising an alarm if the behavioral distance exceeds a threshold, and forwards the response to the client if responses from **L** and **W** are the same.

4.2 Behavioral Distance Between System Call Sequences

We run our experiments on three different `http` server programs: Apache [11], Myserver [1] and Abyss [32]. We choose these servers mainly because they work on both Linux and Windows. A collection of `html` files of size from 0 to 5 MB are served by these `http` servers. Training and testing data is obtained by simulating a client that randomly chooses a file to download. The client sends 1000 requests, out of which 800 are used as training data and the remaining 200 are used as testing data.

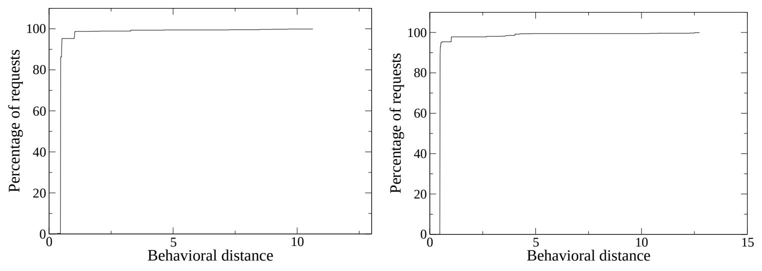
We run two sets of tests. In the first set of tests we run the same server implementation on the replicas, i.e., both **L** and **W** run Apache, Myserver or Abyss. Training data is used to learn the distances between system call phrases, which are then used to calculate the behavioral distance between system call sequences in the testing data. Results of the behavioral distance calculations on the testing data are shown in Figure 3 in the form of cumulative distribution functions (x-axis shows the behavioral distance, and y-axis shows the percentage of requests with behavioral distance smaller than the corresponding value on x-axis.). Figure 3 clearly shows that legitimate requests result in system call sequences with small behavioral distance.

In the second set of tests, we run different servers on **L** and **W**. Figure 4(a) shows the results when **L** is running Myserver and **W** is running Apache, and Figure 4(b) shows results when **L** is running Apache and **W** is running Myserver. Although the behavioral distances calculated are not as small as those obtained in the first set of tests, the results are still very encouraging. This set of tests shows that our system cannot only be used when replicas are running the same servers on different operating systems, but also be used when replicas are running different servers. Our approach is thus an alternative to output voting for server implementations that do not always provide identical responses to the same request (c.f., [4]).



(a) Apache (b) Myserver (c) Abyss

Fig. 3. CDF of behavioral distances when replicas are running the same server



(a) Myserver (**L**) and Apache (**W**) (b) Apache (**L**) and Myserver (**W**)

Fig. 4. CDF of behavioral distances when replicas are running different servers

4.3 Resilience against Mimicry Attacks

Section 4.2 shows that legitimate requests to the replicas result in system call sequences with small behavioral distances. In this section, we show that attack traffic will result in system call sequences of large behavioral distances. However, our emphasis is not on simple attacks which can be detected by intrusion/anomaly detection systems on individual replicas. (We did try two known attacks on an Abyss webserver, and results show that they are detected by isolated anomaly detection systems [37] on any one of the replicas.) Instead, we focus on more sophisticated attacks, namely mimicry attacks [36, 31].

An attack that injects code into the address space of a running process, and then causes the process to jump to the injected code, results in a sequence of system calls issued by the injected code. In a mimicry attack, the injected code is crafted so that the "attack" system calls are embedded within a longer sequence that is consistent with the program that should be running in the process. As shown in [36, 13], mimicry attacks are typically able to evade detection by host-based intrusion/anomaly detection systems that monitor system call sequences.

We analyze a very general mimicry attack, in which the attacker tries to make system call `open` followed by system call `write`, when the vulnerable server is processing a carefully crafted `http` request with attack code embedded. This simple attack sequence is extremely common in many attacks, e.g., the addition of a backdoor root account into the password file. We assume that the attacker can launch such an attack on only one of the replicas using a single request; i.e., either the vulnerability exists only on one of the replicas, or if both replicas are vulnerable, an attacker can inject code that makes system calls of his choice on only one of the replicas. To our knowledge, there is no existing code-injection attacks that violate this assumption, when the replicas are running Linux and Microsoft Windows; nor do we know how to construct one except in very specialized cases.

We perform two tests with different assumptions. The first test assumes that the attacker is trying to evade detection by an existing anomaly detection technique running on one of the replicas. In particular, the anomaly detection technique we consider here is one that uses variable-length system call phrases in modeling normal behavior of the running program [37]. In other words, the first test assumes that the attacker does not know that we are utilizing a behavioral distance calculation between replicas (or indeed that there are multiple replicas). In the second test, we assume that the attacker not only understands that our behavioral distance calculation between replicas is being used, but also has a copy of the distance table that is used in the behavioral distance calculation. This means that an attacker in the second test is the most powerful attacker, who knows everything about our system. In both tests, we exhaustively search for the best mimicry attack. In the first test, the "best" mimicry attack is that which makes the minimal number of system calls while remaining undetected. In the second test, the "best" mimicry attack is that which results in the smallest behavioral distance between system call sequences from the two replicas. We assume that the mimicry attack in both cases results in a request to the uncorrupted replica that produces a "page not found" response.

Results of both tests are shown in Table 1. For each individual test, Table 1 shows the behavioral distance of the best mimicry attack, and the percentage of testing data (from Section 4.2) that has a smaller behavioral distance. That is, the percentage shown in Table 1 indicates the true acceptance rate of our system when the detection threshold is set to detect the best mimicry attack. As shown, these percentages are all very close to 100%, which means that the false alarm rate of our technique is relatively low, even when the system is configured to detect the most sophisticated mimicry attacks. Moreover, by comparing results from the two sets of tests, we can also see the trade-off between better detection capability and lower false positive rate. For example, by setting the threshold to detect any mimicry attacks that could have evaded detection by an isolated intrusion/anomaly detection system on one of the replicas (results in test 1), our system will have a much lower false positive rate (between 0% and 0.5%).

Server on **L**	Apache	Abyss	Myserver	Myserver	Apache
Server on **W**	Apache	Abyss	Myserver	Apache	Myserver
Mimicry on **L** (test 1)	10.283194 99.9093 %	9.821795 100 %	26.656983 100 %	6.908590 99.4555 %	32.764897 100 %
Mimicry on **W** (test 1)	6.842813 99.4555 %	5.492936 99.9093 %	9.967780 99.4555 %	13.354194 100 %	5.280875 99.4555 %
Mimicry on **L** (test 2)	3.736 98.9111 %	1.828 99.8185 %	13.657 100 %	2.731 98.9111 %	13.813 100 %
Mimicry on **W** (test 2)	2.65 98.7296 %	2.687 99.8185 %	2.174 98.0944 %	2.187 98.9111 %	2.64 97.8221 %

Table 1. Behavioral distance of mimicry attacks

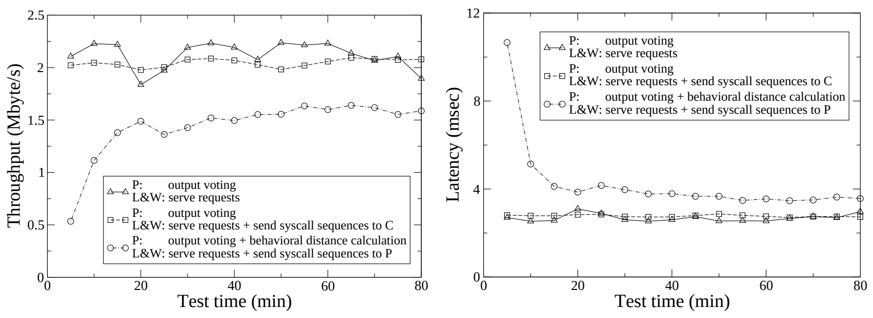
4.4 Performance Overhead

Section 4.2 and Section 4.3 show that our method for behavioral distance is more resilient against mimicry attacks than previous approaches and has low false positive rate. In this section, we evaluate the performance overhead of our implementation of the behavioral distance calculation by measuring the throughput of the `http` servers and the average latency of the requests. The performance evaluation shows that the performance overhead is moderate. Also note that our current implementation is unoptimized, so the performance overhead will be even lower with an optimized implementation.

We run two experiments to evaluate our performance overhead. First, we evaluate the performance degradation of a single server due to the overhead of having to extract and send the system call information to another machine to compute the behavioral distance. Second, we show our performance overhead in comparison to a fault-tolerant system that compares the responses from replicas before returning the response to the client ("output voting").

Performance Overhead of Extracting and Sending System Call Information In this experiment, we run two different tests on one single server running Windows operating system (with a 2.0 GHz Pentium IV processor and 512 MB memory). In both tests, we utilize the static test suite shipped with WebBench 5.0 [34] to test the throughput and latency of the server when the server is fully utilized. In the first test, the machine simply runs the Abyss X1 webserver. In the second test, the machine runs the same webserver and also extracts and sends out the system call information to another machine for the behavioral distance calculation (though this calculation is not on the critical path of the response). We compared the difference in throughput and latency between the two tests. Our experiment results show that the second test has a 6.6% overhead in throughput and 6.4% overhead in latency compared to the first test. This shows that intercepting and sending out system call information causes very low performance overhead on a single server in terms of both throughput and latency.

Performance Overhead Compared to Output Voting We perform three tests to measure the performance overhead of our implementation of the behavioral distance on a replicated system with Abyss X1 webservers. The experimental setup is the same as shown in Section 4.1, except that we use another machine **T** (with a 2.0 GHz Pentium IV processor and 512 MB memory) to generate client requests, and in one of the tests we also have yet another machine **C** to perform the behavioral distance calculation. We use the benchmark program WebBench 5.0 [34] in all the three tests. All tests utilize the static test suite shipped with WebBench 5.0, except that we simulate 10 concurrent clients throughout the tests. Each test was run for 80 minutes with statistics calculated at 5-minute intervals. Results are shown in Figure 5.



(a) Throughput (b) Latency

Fig. 5. Performance overhead

In the first test, replicas **L** and **W** only serve as webservers, without the kernel patch (on Linux) or kernel driver (on Windows) to capture the system call sequences. Proxy **P** does output voting, which means that responses from **L** and **W** are compared before being sent to the client **T**. This test is used as the reference in our evaluation.

In the second test, besides output voting on **P**, replicas **L** and **W** capture the system calls made by the webservers and send them to machine **C**, which does the behavioral distance calculation. Note that in this test the behavioral distance calculation is not on the critical path of responding to the client. The purpose of this test is to show the overhead for capturing the system call information (and analyzing it off-line). As seen from Figure 5, this results in very small overhead: 3.58% in throughput and 0.089 millisecond in latency on average.

In the last test, output voting and the behavioral distance calculation are both performed on the proxy **P** on the critical path of responding to the client, i.e., the response is sent to the client only after the behavioral distance calculation and output comparison complete. To improve performance, **P** caches behavioral distance calculations, so that identical calculations are not performed repeatedly. Figure 5 shows that the proxy needs about 50 minutes to reach its optimal performance level. After that, clients experience about a 24.3% reduction in throughput and 0.848 millisecond overhead in latency, when compared to results from the first test.

The results suggest that we need to use a slightly more powerful machine for the proxy, if we want to do behavioral distance calculation on the critical path of server responses, for servers to remain working at peak throughput. However, even in our tests the overhead in latency is less than a millisecond.

5 Conclusion

In this paper, we introduce behavioral distance for evaluating the extent to which two processes behave similarly in response to a common input. Behavioral distance can be used to detect a software fault or attack on a replica, particularly one that does not immediately yield evidence in the output of the replica. We propose a measure of behavioral distance and a realization of this measure using the system calls emitted by processes. Through an empirical evaluation of this measure using three web servers on two different platforms (Linux and Windows), we demonstrate that this approach is able to detect sophisticated mimicry attacks with low false positive rate and moderate overhead.

References

1. Myserver. `http://www.myserverproject.net`.
2. L. Alvisi, D. Malkhi, E. Pierce, and M. K. Reiter. Fault detection for Byzantine quorum systems. *IEEE Transactions on Parallel Distributed Systems*, 12(9), September 2001.
3. R. W. Buskens and Jr. R. P. Bianchini. Distributed on-line diagnosis in the presence of arbitrary faults. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 470–479, June 1993.
4. M. Castro, R. Rodrigues, and B. Liskov. Base: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3):236–269, 2003.
5. L. Chen and A. Avizienes. n-version programming: A fault-tolerance approach to reliability of software operation. In *Proceedings of the 8th International Symposium on Fault-Tolerant Computing*, pages 3–9, 1978.
6. S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, J. Rowe, S. Staniford-Chen, R. Yip, and D. Zerkle. The design of GrIDS: A graph-based intrusion detection system. Technical Report CSE-99-2, Computer Science Department, U.C. Davis, 1999.
7. C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1998.
8. H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, 2004.

9. H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.
10. S. Forrest and T. A. Langstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.
11. The Apache Software Foundation. Apache http server. `http://httpd.apache.org`.
12. D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graph for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer & Communication Security*, 2004.
13. D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th USENIX Security Symposium*, 2004.
14. J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
15. J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of Symposium on Network and Distributed System Security*, 2004.
16. C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS 2003)*, 2003.
17. L. Lamport. The implementation of reliable distributed multiprocess systems. In *Computer Networks 2*, 1978.
18. X. Lu. A Linux executable editing library. Master's thesis, Computer and Information Science Department, National Unviersity of Singpaore, 1999.
19. G. Nebbett. *Windows NT/2000 Native API Reference*. Sams Publishing, 2000.
20. M. Nei and S. Kumar. *Molecular Evolution and Phylogenetics*. Oxford University Press, 2000.
21. P. Ning, Y. Cui, and D. S. Reeves. Analyzing intensive intrusion alerts via correlation. In *Recent Advances in Intrusion Detection (Lecture Notes in Computer Science vol. 2516)*, 2002.
22. M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.
23. I. Rigoutsos and A. Floratos. Combinatorial pattern discovery in biological sequences. *Bioinformatics*, 14(1):55–67, 1998.
24. T. Romer, G. Voelker, D. Lee, A. Wolman, W.Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of win32/intel executables using etch. In *Proceeding of the USENIX Windows NT Workshop*, August 1997.
25. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
26. B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proceeding of the Working Conference on Reverse Engineering*, pages 45–54, 2002.
27. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.
28. P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26:787–793.
29. K. Shin and P. Ramanathan. Diagnosis of processors with Byzantine faults in a distributed computing system. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 55–60, 1987.

30. S. R. Snapp, S. E. Smaha, D. M. Teal, and T. Grance. The DIDS (Distributed Intrusion Detection System) prototype. In *Proceedings of the Summer USENIX Conference*, pages 227–233, 1992.
31. K. Tan, J. McHugh, and K. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Proceedings of the 5th International Workshop on Information Hiding*, October 2002.
32. Aprelium Technologies. Abyss web server. `http://www.aprelium.com`.
33. A. Valdes and K. Skinner. Probabilistic alert correlation. In *Recent Advances in Intrusion Detection (Lecture Notes in Computer Science vol. 2212)*, 2001.
34. VeriTest. Webbench. `http://www.veritest.com/benchmarks/webbench/default.asp`.
35. D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.
36. D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.
37. A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Proceedings of the 2000 Recent Advances in Intrusion Detection*, 2000.
38. Y. Xie, H. Kim, D. O'Hallaron, M. K. Reiter, and H. Zhang. Seurat: A pointillist approach to anomaly detection. In *Recent Advances in Intrusion Detection (Lecture Notes in Computer Science 3224)*, pages 238–257, September 2004.
39. J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, 2003.