Behavioral Distance Measurement Using Hidden Markov Models

Debin Gao, Michael K. Reiter, and Dawn Song

Carnegie Mellon University
dgao@ece.cmu.edu, reiter@cmu.edu, dawnsong@cmu.edu

Abstract. The *behavioral distance* between two processes is a measure of the deviation of their behaviors. Behavioral distance has been proposed for detecting the compromise of a process, by computing its behavioral distance from another process executed on the same input. Provided that the two processes are diverse and so unlikely to fall prey to the same attacks, an increase in behavioral distance might indicate the compromise of one of them. In this paper we propose a new approach to behavioral distance calculation using a new type of Hidden Markov Model. We also empirically evaluate the intrusion detection capability of our proposal when used to measure the distance between the system-call behaviors of diverse web servers. Our experiments show that it detects intrusions with substantially greater accuracy and with performance overhead comparable to that of prior proposals.

Keywords: intrusion detection, anomaly detection, system call, behavioral distance.

1 Introduction

A predominant form of host-based anomaly detection involves monitoring a process to see if its behavior conforms to the program it is ostensibly executing, e.g., see [15,35,32,28,19,14,17,13,20,16]. Deviation from the behavior prescribed by a program is characteristic of, e.g., code-injection attacks exploiting buffer overflow or format-string vulnerabilities, and so should be investigated. A central research challenge is constructing the model to which the process behavior is compared. This is especially challenging in light of *mimicry* attacks [31,33] on virtually all such models, wherein an adversary injects code that executes its attacks using behaviors that the model does not distinguish from normal.

To better combat mimicry, Gao et al. proposed comparing the behavior of a process to the behavior of another process that is executing on the same input but that either runs on a different operating system or runs a different program that has similar functionality [18]. Assuming their diversity renders these processes vulnerable only to different exploits, a successful attack on one of them should induce a detectable increase in the "distance" between the behaviors of the two processes. In principle, this would make mimicry substantially more difficult, since to avoid detection, the behavior of the compromised process must be close to the simultaneous behavior of the uncompromised one. Gao et al. proposed an approach based on evolutionary distance (ED) [29] to compute behavioral distance, and measured the accuracy and performance of an implementation of this approach when the behavior of each process is the system calls it emits.

In this paper we propose an alternative approach based on a novel Hidden Markov Model (HMM) for computing behavioral distance. An HMM models a doubly stochastic process; there is an underlying stochastic process that is not observable (it is "hidden") but that influences another that produces a sequence of observable symbols. When applied to our problem of computing behavioral distance, the observed symbols are process behaviors (e.g., emitted system calls), and the hidden states correspond to aggregate tasks performed by the processes (e.g., read from a file). Since these hidden tasks should be the same (if the processes are running the same program on different platforms) or at least similar (if the processes are running different programs that offer the same functionality, e.g., two different web servers), it should be possible to reliably correlate the simultaneous observable behaviors of the two processes when no attack is occurring, and to notice an increased behavioral distance when an attack succeeds on one of them. Perhaps surprisingly, our technique uses a single HMM to model both processes simultaneously, in contrast to traditional uses of HMMs for anomaly detection (e.g., [34,10]), where an HMM models a single process.

We detail the distance calculation and model construction algorithms for our HMM-based anomaly detector and evaluate an implementation of it by calculating behavioral distances between processes executing different web servers (Apache[1], Abyss[2], and MyServer[3]) on different platforms (Linux and Windows). Since a significant motivation for this work is constraining mimicry attacks, we also provide an algorithm for estimating the best mimicry against an HMM, and evaluate the false-alarm rate of our approach when the behavioral-distance threshold is set to detect this estimated-best mimicry. In doing so, we show that our approach yields better results than the ED approach of Gao et al., in many cases offering substantial improvement in the false-alarm rate. At the same time, the computational cost is comparable to that of the ED approach in our experiments. As such, we argue that the HMM approach offers substantially superior properties for calculating behavioral distance for anomaly detection.

An alternative strategy to building a behavioral distance measure would be to manually construct a mapping between system calls, or sequences of system calls, on the two platforms of interest. In some cases, such an approach might be aided by the existence of tools such as WINE (`http://www.winehq.com/`), which provides libraries that implement Windows API calls on UNIX to enable the execution of Windows applications on UNIX platforms. For example, an anomaly detector could pattern-match Windows system calls against patterns induced by a call to the Windows API, and then search the Linux system calls for a sequence that corresponds to the WINE implementation of that Windows API call for UNIX. To our knowledge, such an approach has not been studied

[1] `http://httpd.apache.org`
[2] `http://www.aprelium.com`
[3] `http://www.myserverproject.net`

to date, and we eschew it for several reasons. First, we strive for a more general approach that need not be totally reengineered for each new operating system; e.g., we would like an approach that applies with little additional effort to, say, Windows CE and Symbian OS. Second, we want to measure the behavioral distance between even different application codebases (e.g., between the separate codebases of Apache for UNIX and Windows), and we do not expect this manual approach to work well for this case. Third, constructing this mapping manually can be a very substantial effort; e.g., WINE began in 1993 and, at the time of this writing, claims to have UNIX implementations for only 63% of the Windows API (see `http://www.winehq.com/site/winapi_stats`).

Uses of behavioral distance incur the cost of executing each request multiple times. As such, behavioral-distance-based anomaly detection can be most seamlessly integrated into services that already redundantly execute requests for the purposes of detecting (e.g., [30,5,2]) or masking (e.g., [22,27,26,7,6,36,1]) Byzantine faults or intrusions. These approaches ensure that clients receive only correct responses even if a limited number of servers are compromised, by comparing server outputs before they are conveyed to the client ("output voting"). However, a compromised server can do more than simply attempt to mislead a client, e.g., exfiltrating data or attacking other servers, while continuing to provide the proper output to clients. These attacks have typically not been considered in the aforementioned intrusion-tolerant architectures, and since there is already need for servers to be diverse (so as to not fail simultaneously, e.g., see [9,8,11]), these architectures are ripe for the integration of behavioral-distance-based anomaly detection to augment the protections they provide.

2 Related Work

Behavioral-distance-based anomaly detection is most closely related to the recent work of Cox et al. on *N-variant systems* [11]. In an N-variant system, the behaviors of multiple processes on a common input are compared to detect deviations, as in the framework we consider here. The focus in N-variant systems, however, is to construct these multiple processes through mechanical transformation so that necessary conditions for a certain type of attack cannot be satisfied in all processes. For example, if two processes are created to execute the same program but with disjoint address spaces (i.e., an address valid in one is necessarily invalid in the other), then an attack that depends on accessing an absolute address will crash at least one of the processes. Cox et al. anticipate the use of a *monitor* to detect attacks other than by output voting, though to our knowledge they have not explored monitoring behavior at the system-call level or via any technique as general as the approach we describe here. Another difference is that the N-variant system usually requires a special compiler or a binary rewriter to construct a variant, whereas our approach is a black-box approach which does not require source code or static analysis of the binary.

Another technique proposed to make mimicry attacks more difficult utilizes system-call arguments (e.g., [21,4]). Models for detecting anomalous system calls typically monitor the system-call numbers but not their arguments, and so a mimicry attack can issue system calls that are consistent with the model but for which the arguments of certain calls are modified to be "malicious". To the extent that system-call arguments can be accurately modeled, this can increase the difficulty of mimicry attacks. While we do not utilize system-call arguments in this work, it is potentially a way to augment the strength of our technique.

The key to the technique we present here is a novel HMM construction. HMMs have been studied for decades and used in a wide variety of applications, owing to two features: First, HMMs are very rich in mathematical structure and hence can form the theoretical basis for a wide range of applications. Second, when applied properly, HMMs work very well in practice for many important applications. One of the most successful applications of HMMs is in speech recognition [25]. HMMs have also been used in intrusion detection systems, e.g., to model the system-call behavior of a single process [34], and to model privilege flows [10]. However, these HMMs are designed to model the behavior of a single process, as opposed to the joint behavior of two processes as we require here.

Variations of ordinary HMMs might seem to be more suited to our needs. For example, "pair HMMs" [23] and "generalized pair HMMs" [24] have been used to model joint distributions, specifically to predict the gene structures of two unannotated input DNA sequences. However, these variations of HMMs only model two observable sequences where symbols are drawn from the same alphabet. In our case, not only are the alphabets—i.e., the system calls on diverse platforms—different, but the correspondences between these alphabets are not known and are not one-to-one. As such, we have been unable to directly adapt these prior techniques to our problem, and have devised a custom solution, instead.

3 Motivation for Our Approach

In a nutshell, the problem is to assign a distance to a pair of system call sequences

$$S_1 = \langle s_{1,1}, s_{1,2}, \ldots, s_{1,l_1} \rangle \qquad S_2 = \langle s_{2,1}, s_{2,2}, \ldots, s_{2,l_2} \rangle \tag{1}$$

emitted by two processes while processing the same input. Here, each $s_{i,j}$ denotes the system-call number (a natural number) of the j-th system call by the i-th process. The distance should indicate whether these sequences reflect similar activities. Producing this distance is complicated by the fact that the processes might be running on diverse platforms, and so the set of system calls $C_1 = \{s_{1,j}\}_{1 \le j \le l_1}$ on the first platform can be different from the set $C_2 = \{s_{2,j}\}_{1 \le j \le l_2}$ on the second platform. Moreover, even a shared symbol $c \in C_1 \cap C_2$ has different semantics on the two platforms. Of course, generally $l_1 \neq l_2$.

The evolutionary distance (ED) approach [18] to computing the distance of (1), roughly speaking, was to consider all possible ways of inserting dummy symbols σ into them to generate an *alignment*

$$\langle s'_{1,1}, s'_{1,2}, \ldots, s'_{1,l'_1} \rangle \qquad \langle s'_{2,1}, s'_{2,2}, \ldots, s'_{2,l'_2} \rangle \tag{2}$$

where $l'_1 \ge l_1$, $l'_2 \ge l_2$, and $l'_1 = l'_2$. The distance for alignment (2) was simply $\sum_j \mathsf{dist}(s'_{1,j}, s'_{2,j})$, where dist was a table of distances between system calls learned from training sequences (pairs of system call sequences output by the processes in a benign environment). The distance for (1), then, was the distance of the alignment with the smallest distance.

Though we have omitted numerous details of the ED approach, one limitation is immediately apparent: it does not take adequate account of the order of system calls in each sequence. For example, reversing the two sequences (1) yields the same behavioral distance. Since system-call order is known to be important to detecting intrusions (e.g., [15,28,17,16]), this is a significant limitation.

Our use of an HMM for calculating the behavioral distance of sequences (1) addresses this limitation. We use a single HMM to model both processes, and so a pair of system calls $[s_{1,\cdot}, s_{2,\cdot}]$, one from each process, is an observable symbol of the HMM. Each such observable symbol can be emitted by hidden states of the HMM with some finite probability. Intuitively, if the system calls in an observable symbol perform similar tasks, then the probability should be high, otherwise the probability should be low. This probability serves the same purpose as the dist table in the ED approach. However, in HMM-based behavioral distance, the probability of emitting the same observable symbol is generally different for different states, whereas in ED-based behavioral distance, a universal dist table is used for every system call pair in the system call sequences. In this way, our HMM model better accounts for the order of system calls.

The way in which we use our HMM is slightly different from HMM use in many other applications. For example, in HMM-based speech recognition, the primary algorithmic challenge is to find the most probable state sequence (what is being said) given the observable symbol sequence (the recorded sounds). However, in behavioral distance, we are not concerned about the tasks (the hidden states) that gave rise to the observed system call sequences, but rather are concerned only that they match. Therefore, the main HMM problem we need to solve is to determine the probability with which the given system call sequences would be generated (together) by the HMM model—we take this probability as our measure of the behavioral distance. We show how to calculate this probability efficiently in Section 4.

4 The Hidden Markov Model

In this section, we introduce our Hidden Markov Model and describe how it is used for behavioral distance calculation. We begin in Section 4.1 with an overview of the HMM. We then present our algorithm for calculating the behavioral distance in Section 4.2, and describe the original construction of the HMM in Section 4.3.

4.1 Elements of the HMM

Our HMM $\lambda = (Q, V, A, B)$ consists of the following components:

– A set $Q = \{q_0, q_1, q_2, \ldots, q_N, q_{N+1}\}$ of states, where q_0 is a designated *start state*, and q_{N+1} is a designated *end state*.

- A set $V = \{[x, y] : x \in C_1 \cup \{\sigma\}, y \in C_2 \cup \{\sigma\}\}$ of output symbols. Recall that C_1 and C_2 are the sets of system calls[4] observed on platforms 1 and 2, respectively, and that σ denotes a designated dummy symbol.
- A set $A = \{a_i\}_{0 \leq i \leq N}$ of state transition probability distributions. Each $a_i : \{1, \ldots, N+1\} \rightarrow [0, 1]$ satisfies $\sum_j a_i(j) = 1$. $a_i(j)$ is the probability that the HMM, when in state q_i, will next enter q_j. We will typically denote $a_i(j)$ with $a_{i,j}$. We stipulate that $a_{0,N+1} = 0$, i.e., the HMM does not transition directly from the start state to the end state. Note that a_i is undefined for $i = N + 1$, i.e., there are no transitions from the end state. Similarly, $a_{i,0}$ is undefined for all i, since there are no transitions to the start state.
- A set $B = \{b_i\}_{1 \leq i \leq N}$ of symbol emission probability distributions. Each $b_i : (C_1 \cup \{\sigma\}) \times (C_2 \cup \{\sigma\}) \rightarrow [0, 1]$ satisfies $\sum_{[x,y]} b_i([x, y]) = 1$. $b_i([x, y])$ is the probability of the HMM emitting $[x, y]$ when in state q_i. We require that for all i, $b_i([\sigma, \sigma]) = 0$. Note that neither b_0 nor b_{N+1} is defined, i.e., the start and end states do not emit symbols.

As we discussed in Section 3, we will take our measure of behavioral distance to be the probability with which the HMM λ "generates" the pair of system call sequences of interest. This probability is computed with respect to the following experiment, which we refer to as "executing" the HMM:

1. Initialize λ with q_0 as the current state.
2. Repeat the following until q_{N+1} is the current state:
   (a) If q_i is the current state, then select a new state q_j according to the probability distribution a_i and assign q_j to be the new current state.
   (b) After transitioning to the new state q_j, if $q_j \neq q_{N+1}$ then select an output symbol $[x, y]$ according to the probability distribution b_j and emit it.

Specifically, we define an *execution* π of the HMM λ to consist of a state sequence $q_{i_0}, q_{i_1}, \ldots, q_{i_T}$, where $i_0 = 0$ and $i_T = N + 1$, and observable symbols $[x_{i_1}, y_{i_1}], \ldots, [x_{i_{T-1}}, y_{i_{T-1}}]$. The experiment above assigns to each execution a probability, i.e., the probability the experiment traverses exactly that sequence of states and emits exactly that sequence of observable symbols; we denote by $\Pr_\lambda(\pi)$ the probability of execution π when executing HMM λ.

For the HMM λ we will build, there are many executions that generate the given pair of sequences $[S_1, S_2]$ as in (1). We use $\mathsf{Ex}_\lambda([S_1, S_2])$ to denote the set of executions of λ that generate $[S_1, S_2]$. The probability that λ generates the sequences $[S_1, S_2]$ in (1), which we denote $\Pr_\lambda([S_1, S_2])$, is the probability that λ, in the experiment above, emits pairs $[x_{i_1}, y_{i_1}], \ldots, [x_{i_{T-1}}, y_{i_{T-1}}]$ such that

$$\langle x_{i_1}, x_{i_2}, \ldots, x_{i_{T-1}} \rangle \qquad \langle y_{i_1}, y_{i_2}, \ldots, y_{i_{T-1}} \rangle$$

is an alignment (as in (2)) of those sequences. Note that

$$\Pr_\lambda([S_1, S_2]) = \sum_{\pi \in \mathsf{Ex}_\lambda([S_1, S_2])} \Pr_\lambda(\pi)$$

[4] In Section 4.4, we discuss letting C_1 and C_2 be sets of system call sequences, or *phrases*. For simplicity of exposition, however, we describe our algorithms assuming C_1 and C_2 are sets of individual system calls.

In addition, we define the *most probable execution* generating $[S_1, S_2]$ to be

$$\arg\max_{\pi \in \mathsf{Ex}_\lambda([S_1,S_2])} \Pr_\lambda(\pi)$$

When convenient, we will use t to denote an iteration counter, i.e., the number of iterations of Step 2 in the experiment above that have been executed. So, for example, when we say that λ is "in state q_i after t iterations", this means that after t iterations have been completed in the experiment, q_i is the current state. Trivially, q_0 is the state after $t = 0$ iterations, and if the state is q_{N+1} after t iterations, then execution halts (i.e., there is no iteration $t + 1$).

4.2 Computing $\mathbf{Pr}_\lambda([S_1, S_2])$

$\Pr_\lambda([S_1, S_2])$ is the probability that system call sequences S_1 and S_2 are generated (in the sense of Section 4.1) by the HMM λ, which is used as the behavioral distance between S_1 and S_2. If $\Pr_\lambda([S_1, S_2])$ is greater than a threshold value, the system call sequences will be considered as normal, otherwise an alarm is raised indicating that an anomaly is detected. In this section we describe an algorithm for computing $\Pr_\lambda([S_1, S_2])$ efficiently, given λ, S_1, and S_2. Again, S_1 and S_2 would typically be observed from monitoring the processes. How we build λ itself is the topic of Section 4.3.

Given an HMM λ, there are many ways it can generate S_1 and S_2, i.e., there are many different executions that yield an alignment of S_1 and S_2. In fact, if we assume that $a_{i,j}$ and $b_i([x, y])$ are non-zero for $x \neq \sigma$ or $y \neq \sigma$, any state sequence of sufficient length generates an alignment of S_1 and S_2 with some non-zero probability. Moreover, even for one particular state sequence, there are many ways of generating S_1 and S_2 with σ inserted at different locations.

It may first seem that to calculate $\Pr_\lambda([S_1, S_2])$ we need to sum the probabilities of all possible executions, and the large number of executions makes the algorithm very inefficient. However, we can use induction to find $\Pr_\lambda([S_1, S_2])$, instead. The idea is that if we know the probability of generating $[S_1^-, S_2^-]$, where S_1^- and S_2^- are prefixes of S_1 and S_2, respectively, then $\Pr_\lambda([S_1, S_2])$ can be found by extending the executions that generate S_1^- and S_2^-.

To express this algorithm precisely, we introduce the following random variables in an execution of the HMM λ. Random variable State^t is the state after t iterations. (It is undefined if the execution terminates in less than t iterations.) Random variable $\mathsf{Out}_1^{\leq t}$ is the sequence of system calls from C_1 in the first components of the emitted symbols (less σ) through t iterations. That is, if in the (up to) t iterations, λ emits $[s'_{1,1}, s'_{2,1}], \ldots, [s'_{1,\ell}, s'_{2,\ell}]$ where $\ell \leq t$, then $\mathsf{Out}_1^{\leq t}$ is the sequence of non-σ values in $\langle s'_{1,1}, \ldots, s'_{1,\ell}\rangle$ (with their order preserved). Similarly, the random variable $\mathsf{Out}_2^{\leq t}$ would be the non-σ values in $\langle s'_{2,1}, \ldots, s'_{2,\ell}\rangle$. Now define

$$\alpha(u, v, i) = \Pr_\lambda\left(\bigvee_{t \geq 0}\left(\mathsf{State}^t = q_i \wedge \mathsf{Out}_1^{\leq t} = \mathrm{Pre}(S_1, u) \wedge \mathsf{Out}_2^{\leq t} = \mathrm{Pre}(S_2, v)\right)\right)$$

where $\mathrm{Pre}(S, u)$ denotes the u-length prefix of S. That is, $\alpha(u, v, i)$ is the probability of the event that simultaneously q_i is the current state, exactly the first u system calls for process 1 have been emitted, and exactly the first v system calls for process 2 have been emitted. Clearly $\alpha(u, v, i)$ is a function of S_1, S_2, and λ. Here we do not specify them as long as the context is clear. We solve for $\alpha(u, v, i)$ inductively, as follows.

Base cases:

$$\alpha(0,0,i) = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases} \qquad \alpha(u,v,0) = \begin{cases} 1 & \text{if } u = v = 0 \\ 0 & \text{otherwise} \end{cases}$$

Induction:

$$\alpha(u,0,i) = \sum_{j=0}^{N} \alpha(u-1,0,j) a_{j,i} b_i([s_{1,u}, \sigma]) \qquad \text{for } u > 0, i > 0$$

$$\alpha(0,v,i) = \sum_{j=0}^{N} \alpha(0,v-1,j) a_{j,i} b_i([\sigma, s_{2,v}]) \qquad \text{for } v > 0, i > 0$$

$$\begin{aligned}\alpha(u,v,i) = &\sum_{j=0}^{N} \alpha(u-1,v,j) a_{j,i} b_i([s_{1,u}, \sigma]) + \sum_{j=0}^{N} \alpha(u,v-1,j) a_{j,i} b_i([\sigma, s_{2,v}]) \\ &+ \sum_{j=0}^{N} \alpha(u-1,v-1,j) a_{j,i} b_i([s_{1,u}, s_{2,v}]) \qquad \text{for } u, v > 0, i > 0\end{aligned}$$

For example, $\alpha(1, 0, i)$ is the probability that q_i is the current state and all that has been emitted is one system call for process 1 ($s_{1,1}$) and nothing (except σ) for process 2. Since $b_j([\sigma, \sigma]) = 0$ for all $j \in \{1, \dots, N\}$, the only possibility is that q_0 transitioned directly to q_i, which emitted $[s_{1,1}, \sigma]$.

As a second example, to solve for $\alpha(u, v, i)$ where $u, v > 0$, there are three possibilities, captured in the last equation above:

– The first $u - 1$ and v system calls from S_1 and S_2, respectively, have been output, and λ is in some state q_j. (This event occurs with probability $\alpha(u-1, v, j)$.) λ then transitions from q_j to q_i (with probability $a_{j,i}$) and emits $[s_{1,u}, \sigma]$ (with probability $b_i([s_{1,u}, \sigma])$).
– The first u and $v - 1$ system calls from S_1 and S_2, respectively, have been output, and λ is in some state q_j. (This event occurs with probability $\alpha(u, v-1, j)$.) λ then transitions from q_j to q_i (with probability $a_{j,i}$) and emits $[\sigma, s_{2,v}]$ (with probability $b_i([\sigma, s_{2,v}])$).
– The first $u - 1$ and $v - 1$ system calls from S_1 and S_2, respectively, have been output, and λ is in some state q_j. (This event occurs with probability $\alpha(u-1, v-1, j)$.) λ then transitions from q_j to q_i (with probability $a_{j,i}$) and emits $[s_{1,u}, s_{2,v}]$ (with probability $b_i([s_{1,u}, s_{2,v}])$).

After $\alpha(u, v, i)$ is solved for all values of $u \in \{0, 1, \dots, l_1\}$, $v \in \{0, 1, \dots, l_2\}$, and $i \in \{1, \dots, N\}$, where l_1 and l_2 are the lengths of S_1 and S_2, respectively, we can calculate

$$\Pr_{\lambda}([S_1, S_2]) = \sum_{i=1}^{N} \alpha(l_1, l_2, i) a_{i,N+1}$$

The solution above solves for $\Pr_{\lambda}([S_1, S_2])$ from the beginning of the system call sequences. (That is, $\alpha(u, v, i)$ of smaller u- and v-indices are found before that of larger u- and v-indices.) It will also be convenient to solve for $\Pr_{\lambda}([S_1, S_2])$ from the end of the sequences. To do that, we define

$$\beta(u, v, i) = \Pr_{\lambda}\left(\bigvee_{t \geq 0} \left(\mathsf{State}^t = q_i \wedge \mathsf{Out}_1^{>t} = \mathrm{Post}(S_1, u) \wedge \mathsf{Out}_2^{>t} = \mathrm{Post}(S_2, v)\right)\right)$$

Here, $\mathrm{Post}(S, u)$ denotes the suffix of S that remains after removing the first u elements of S. Analogous to the preceding discussion, random variable $\mathsf{Out}_1^{>t}$ is the sequence of system calls from C_1 in the first components of the emitted symbols (less σ) in iterations $t+1$ onward (if any), and similarly for $\mathsf{Out}_2^{>t}$. So, $\beta(u, v, i)$ is the probability of the event that q_i is the current state after some iterations and subsequently exactly the last $l_1 - u$ system calls of S_1 are emitted, and exactly the last $l_2 - v$ system calls of S_2 are emitted. The induction for $\beta(u, v, i)$ works in a similar way, and $\Pr_{\lambda}([S_1, S_2]) = \beta(0, 0, 0)$.

In this algorithm, the number of steps taken to calculate $\Pr_{\lambda}([S_1, S_2])$ is proportional to $l_1 \times l_2 \times N^2$. Therefore, the proposed algorithm is efficient as the numbers of system calls and HMM states grow.

4.3 Building λ

In this section we describe how we build the HMM λ. We do so using training data, that is, pairs $[S_1, S_2]$ of sequences of system calls recorded from the two processes when processing the same inputs. Of course, we assume that these training pairs reflect only benign behavior, and that neither process is compromised during the collection of the training samples. We first present an algorithm to adjust the HMM parameters for one training example $[S_1, S_2]$, and then show how we combine the results from processing each training sample to adjust the HMM when there are multiple training samples.

Building λ is a typical expectation-maximization problem. There is no known way of solving for such a maximum likelihood model analytically; therefore a refinement procedure is used. The idea is that for each training sample $[S_1, S_2]$, we find the expected values of certain variables, which can, in turn, be used to adjust the parameters of λ to increase $\Pr_{\lambda}([S_1, S_2])$. Here we will demonstrate this method for updating the a_i parameters of λ; a similar treatment for the b_i parameters can be found in Appendix A.

The initial instance of λ is created with a fixed number of states N and random a_i and b_i distributions. To update the $a_{i,j}$ parameters in light of a training sample $[S_1, S_2]$, we find (for the current instance of λ) the expected number of times λ transitions to state q_i when generating $[S_1, S_2]$, and the expected number of times it transitions from q_i to q_j when generating $[S_1, S_2]$. To compute these expectations, we first define two conditional probabilities, $\gamma(u,v,i)$ and $\xi(u,v,i,j)$ for $i \leq N, j \leq N+1$, as follows:

$$\gamma(u,v,i) = \mathrm{Pr}_\lambda\left(\left(\bigvee_{t\geq 0} \begin{array}{l} \mathsf{State}^t = q_i \ \wedge \\ \mathsf{Out}_1^{\leq t} = \mathrm{Pre}(S_1,u) \ \wedge \\ \mathsf{Out}_2^{\leq t} = \mathrm{Pre}(S_2,v) \end{array}\right) \middle| \begin{pmatrix} \mathsf{Out}_1^{>0} = S_1 \ \wedge \\ \mathsf{Out}_2^{>0} = S_2 \end{pmatrix}\right)$$

$$\xi(u,v,i,j) = \mathrm{Pr}_\lambda\left(\left(\bigvee_{t\geq 0} \begin{array}{l} \mathsf{State}^t = q_i \ \wedge \ \mathsf{State}^{t+1} = q_j \ \wedge \\ \mathsf{Out}_1^{\leq t} = \mathrm{Pre}(S_1,u) \ \wedge \\ \mathsf{Out}_2^{\leq t} = \mathrm{Pre}(S_2,v) \end{array}\right) \middle| \begin{pmatrix} \mathsf{Out}_1^{>0} = S_1 \ \wedge \\ \mathsf{Out}_2^{>0} = S_2 \end{pmatrix}\right)$$

That is, $\gamma(u,v,i)$ is the probability of λ being in state q_i after emitting u system calls for process 1 and v system calls for process 2, given that the entire sequences for process 1 and process 2 are S_1 and S_2, respectively. Similarly, $\xi(u,v,i,j)$ is the probability of being in state q_i after emitting u system calls for process 1 and v system calls for process 2, and then transitioning to state q_j, given the entire system call sequences for the processes. Each of these conditional probabilities pertains to one particular subset of executions that generate S_1 and S_2. As explained in Section 4.2, there are many executions in the HMM that are able to generate S_1 and S_2; out of these executions, there are some that are in state q_i (respectively, transition from q_i to q_j) after emitting u system calls for process 1 and v system calls for process 2. Note that it may or may not be the case that $[s_{1,u}, s_{2,v}]$ was emitted by state q_i, and that

$$\gamma(u,v,i) = \sum_{j=1}^{N+1} \xi(u,v,i,j)$$

We can calculate these quantities easily as follows:

$$\gamma(u,v,i) = \frac{\alpha(u,v,i)\beta(u,v,i)}{\mathrm{Pr}_\lambda([S_1,S_2])}$$
$$\xi(u,v,i,j) = \frac{1}{\mathrm{Pr}_\lambda([S_1,S_2])}\begin{pmatrix} \alpha(u,v,i)a_{i,j}b_j([s_{1,u+1},\sigma])\beta(u+1,v,j) + \\ \alpha(u,v,i)a_{i,j}b_j([\sigma,s_{2,v+1}])\beta(u,v+1,j) + \\ \alpha(u,v,i)a_{i,j}b_j([s_{1,u+1},s_{2,v+1}])\beta(u+1,v+1,j) \end{pmatrix}$$

Let the random variable X_i be the number of times that state q_i is visited when emitting $[S_1,S_2]$. We calculate the expected value of X_i, denoted $\mathbb{E}(X_i)$, as follows. Let the random variable $X_i^{u,v}$ be the number of times that q_i is the current state when exactly the first u system calls of S_1 and the first v system calls of S_2 have been emitted. Since q_i can be visited at most once for a fixed u and v, $X_i^{u,v}$ can take on only values 0 and 1. As such, $\mathbb{E}(X_i^{u,v}) = \sum_{x\in\{0,1\}} x\mathrm{Pr}(X_i^{u,v} = x) = \gamma(u,v,i)$. Then, by linearity of expectation,

$$\mathbb{E}(X_i) \ = \ \sum_{u=0}^{l_1}\sum_{v=0}^{l_2}\mathbb{E}(X_i^{u,v}) \ = \ \sum_{u=0}^{l_1}\sum_{v=0}^{l_2}\gamma(u,v,i)$$

where l_1 and l_2 are the lengths of S_1 and S_2, respectively. Similarly, if $X_{i,j}$ is the number of transitions from q_i to q_j when generating $[S_1, S_2]$, then

$$\mathbb{E}(X_{i,j}) \;=\; \sum_{u=0}^{l_1}\sum_{v=0}^{l_2} \xi(u, v, i, j)$$

With these expectations calculated, we can update the a_i parameters of the HMM λ, using the Baum-Welch method [3], as follows:

$$a_{i,j} \leftarrow \mathbb{E}(X_{i,j})/\mathbb{E}(X_i)$$

These equations show how the a_i parameters of λ can be updated to increase the probability of generating one pair of sequences. When there are more than one pair of sequences ($[S_1^{(1)}, S_2^{(1)}]$, …, $[S_1^{(M)}, S_2^{(M)}]$), the above equations can be used to calculate the relevant parameters for each pair of sequences (i.e., $\mathbb{E}(X_i^{(k)})$, $\mathbb{E}(X_{i,j}^{(k)})$) and then the a_i parameters of λ can be updated as

$$a_{i,j} \leftarrow \left(\sum_{k=1}^{M} w_k \mathbb{E}(X_{i,j}^{(k)})\right) \Big/ \left(\sum_{k=1}^{M} w_k \mathbb{E}(X_i^{(k)})\right)$$

where w_k is the weight for each pair of sequences $[S_1^{(k)}, S_2^{(k)}]$ in the training set for the current instance of λ. There are many ways of setting w_k [12]. In our experience, different settings affect the speed of convergence, but the final result of the HMM is almost the same. In our experiments, we choose

$$w_k = \left(\Pr_\lambda([S_1^{(k)}, S_2^{(k)}])\right)^{-\frac{1}{l_1^{(k)}+l_2^{(k)}}}$$

where $l_1^{(k)}$ and $l_2^{(k)}$ are the lengths of $S_1^{(k)}$ and $S_2^{(k)}$, respectively.

The equations above show how the parameters of an HMM can be adjusted in one refinement. We need many such refinements in order to find a good HMM that generates the training examples with high probabilities. Although more refinements can improve the probabilities, they may also result in overfitting. To detect when to stop the refinement process so as not to overfit the training samples, we use a separate validation set, which also contains pairs of system call sequences recorded from the two processes when processing the same inputs. Briefly, we detect overfitting when the refinement process either decreases $\Pr_\lambda([S_1, S_2])$ for pairs $[S_1, S_2]$ in the validation set or increases the false-alarm rate on the validation set using the alarm threshold needed to detect mimicry attacks (explained in Section 5.1).

4.4 Implementation Issues

There are several implementation issues that deserve comment. First, in all discussion so far, we have used system calls as the basic units to explain the elements of the HMM and our algorithms; i.e., an observable symbol of the HMM is a pair of system calls, one from each process. However, it is advantageous to use system call *phrases* (short sequences of system calls) as the basic unit [35,17,18]. In our experiments, we use the same phrase-extraction algorithm as in the ED project [18]. After the system call phrases are identified, an observable symbol of the HMM becomes a pair of system call phrases, one from each process. Other than this, all algorithms presented in this paper remain the same.

Second, the number N of states in the HMM must be set before training starts. (N does not change once it is set.) A small N will make the HMM not as powerful as required to model the behavior of the processes, which will, in turn, make mimicry attacks relatively easy. However, a large N not only degrades the performance of the system, but may also result in overfitting the training data. We have found success in setting N slightly larger than the length of the longest training sequence so that some dummy symbols σ can be inserted into the sequences, and to use the validation set to detect overfitting. So far we have found that setting N to be 1.0 to 1.2 times the length of the longest training sequence (in phrases) is a reasonable guideline. In our experiments described in Section 5 using three different web servers on two different operating systems, this guideline yielded values of N between 10 and 33.

Third, the training of the HMM is a complicated process, which may take a long time. In our experiments, the training for a typical web server application may take more than an hour on a desktop computer with a Pentium IV 3.0 GHz CPU. However, training can be performed offline, and the online monitoring is fast, as in many other applications of HMMs.

A fourth issue concerns the use of a finite set of training samples for estimating the HMM parameters. If we look at the formulas for building the HMM in Section 4.3, we see that certain parameters will be set to 0 if there are no or few occurrences of a symbol in the training set. For example, if an observable symbol does not occur often enough, then the probability of that symbol being emitted will be 0 in some states. This should be avoided because no occurrences in the training data might be the result only of a low, but still nonzero, probability of that event. Therefore, in our implementation we ensure a (nonzero) minimum value to the a_i and b_i parameters by adding a normalization step at the end of each refinement process.

5 Evaluation and Discussion

As discussed in Section 4, we hypothesized that because the HMM-based approach we advocate here better accounts for the order of system calls, it should better defend against mimicry attacks than the prior ED-based approach [18]. In this section, we evaluate an implementation of our anomaly detector using HMM-based behavioral distance to determine whether this is, in fact, true, and to gain insight into the computational cost of our approach.

Our evaluation system includes two computers running web servers to process client HTTP requests. One of these computers, denoted **L**, runs Linux kernel 2.6.8, and the other, denoted **W**, runs Windows XP Pro SP2. The web server run by each computer differs from test to test, and will be discussed below. In our tests, each of **L** and **W** was given the same sequence of requests (generated from the static test suite of WebBench 5.0,[5] and each recorded the system call sequence, denoted by $S_{\mathbf{L}}$ and $S_{\mathbf{W}}$,[6] respectively, of (the thread in) the web server process that handled the request. The behavioral distance is calculated as $\Pr_\lambda([S_{\mathbf{L}}, S_{\mathbf{W}}])$, where λ was trained as described in Section 4.3.

5.1 Resilience Against Mimicry Attacks

Our chosen measure of the system's resilience to mimicry attacks is the false-alarm rate of the system when it is configured to detect the "best" mimicry attack. Intuitively, a system that offers a low false-alarm rate while detecting the best mimicry attack is doing a good job of discriminating "normal" behavior from even the "best-disguised" abnormal behavior. To compare our results to the ED-based behavioral distance project [18], we presume the same system call sequence that the attacker is trying to execute as in the ED project, which is simply an `open` followed by a `write`.

To measure the false-alarm rate when detecting the best mimicry, we need to first define what we take as the "best" mimicry attack. Specifically, if we presume that the attacker finds a vulnerability in, say, **L**, then it must craft an attack request that will produce a "normal" behavioral distance between the attack activity on **L** induced by its request ($S_{\mathbf{L}}$) and the normal activity on **W** induced by the same request ($S_{\mathbf{W}}$). Moreover, the attack activity on **L** must include an `open` followed by a `write` (i.e., the attacker's system calls). As such, it would be natural to define the "best" mimicry attack to be the one that yields the most normal behavioral distance, i.e., that maximizes $\Pr_\lambda([S_{\mathbf{L}}, S_{\mathbf{W}}])$. Because we permit the attacker to have complete knowledge of our HMM λ, nothing is hidden from the attacker to prevent his use of this "best" mimicry attack.

Unfortunately, we know of no efficient algorithm for finding this best mimicry attack (an obstacle an attacker would also face), and so we have to instead evaluate our system using an "estimated-best" mimicry attack that we can find efficiently. Rather than maximizing $\Pr_\lambda([S_{\mathbf{L}}, S_{\mathbf{W}}])$, this estimated-best mimicry attack is the one produced by the most probable execution of the HMM λ that includes the attacker's system calls on the platform we presume he can compromise. (The most probable execution does not necessarily yield the mimicry attack that maximizes $\Pr_\lambda([S_{\mathbf{L}}, S_{\mathbf{W}}])$, since many low-probability executions can yield a different $[S'_{\mathbf{L}}, S'_{\mathbf{W}}]$ that has a larger $\Pr_\lambda([S'_{\mathbf{L}}, S'_{\mathbf{W}}])$.) An algorithm for computing this estimated-best mimicry attack can be found in Appendix B. Another way in which our attack is "estimated-best" is that it assumes the attacker executes its attack within the servers' processing of a single request (an assumption made in the ED project [18] as well). Attacks for which the attack activity spans multiple requests or multiple server processes/threads is an area of ongoing work.

[5] VeriTest, `http://www.veritest.com/benchmarks/webbench/default.asp`

[6] System calls on Windows are also called native API calls or kernel calls. We obtain the Windows system call information by overwriting the KiSystemService table in the Windows kernel using a kernel driver we developed.

Once this estimated-best mimicry attack is found, we set the behavioral distance alarm threshold to be the behavioral distance resulting from this estimated-best mimicry, and measure the false-alarm rate of the system that results. A false alarm corresponds to a legitimate request that induces a pair of system call sequences with a probability of emission from λ at most the threshold. The false-alarm rate is then calculated as the number of false alarms divided by the total number of requests. We perform our experiments in nine different settings, defined by the web servers that **L** and **W** are running. (The web servers are Apache 2.0.54, Abyss X1 2.0.6 and MyServer 0.8.) Table 1 presents results using a testing mechanism in which the training (to train the model), validation (to detect overfitting) and evaluation (to evaluate) sets are distinct. They show that the HMM-based behavioral distance has a small (and in many cases, greatly superior to ED) false-alarm rate when detecting the estimated-best mimicry attacks.

Table 1. False-alarm rate when detecting the estimated-best mimicry attack

Server on **L**	Server on **W**	ED-based Mimicry on **L**	ED-based Mimicry on **W**	HMM-based Mimicry on **L**	HMM-based Mimicry on **W**
Apache	Apache	2.08 %	0.16 %	0 %	0.16 %
Abyss	Abyss	0.4 %	0.32 %	0.16 %	0.08 %
MyServer	MyServer	1.36 %	1.2 %	0 %	0 %
Apache	Abyss	0.4 %	0.32 %	0 %	0.16 %
Abyss	Apache	0.8 %	0.48 %	0.08 %	0.08 %
Apache	MyServer	0 %	3.65 %	0 %	0 %
MyServer	Apache	6.4 %	0.16 %	0 %	0 %
Abyss	MyServer	0 %	1.91 %	0 %	1.44 %
MyServer	Abyss	0.4 %	0.08 %	0.4 %	0 %

5.2 Performance Overhead

To evaluate the performance overhead of a system using our HMM-based behavioral distance, we run two experiments. First, we measure the time it takes to calculate the behavioral distance, and compare that with the ED-based approach. Second, we apply the HMM-based behavioral distance on real servers and evaluate its performance overhead.

In the first experiment, we measure the time it takes for our implementations of the behavioral distance measurement (both the ED-based and the HMM-based) to calculate the behavioral distance of 1200 pairs of system call sequences on a Pentium IV 2.0GHz computer with 512MB of memory. In 10 runs of the experiment, the HMM-based calculation takes 2.269 seconds on average, and the ED-based calculation takes 2.422 seconds on average. As such, our HMM-based calculation is 6.32% faster than the ED-based calculation.

In the second experiment, we augment the setup containing **L** and **W** with two additional machines, a proxy **P** and a client **C**, and connect them in a 100 Mbps local area network. Table 2 summarizes the properties of **L**, **W**, **P**, and **C**. The client **C** submits requests to the proxy **P**, which forwards the requests to both **L** and **W** for processing. Responses from **L** and **W** are sent to **P**, which then sends a response to **C**. **C** uses the benchmark program WebBench 5.0 to issue requests. All tests utilize the static test suite shipped with WebBench 5.0, with a setting of 10 concurrent client threads. Each test was run for 1600 seconds with statistics calculated at 100-second intervals. In these tests, both **L** and **W** run the Apache web server 2.2.2.

Table 2. Configurations of computers in the performance overhead evaluation

Machine Name	Operating System	CPU	Memory	Remarks
L	Linux kernel 2.6.8	Pentium IV 2.0 GHz	512 MB	Replica
W	Windows XP Pro SP2	Pentium IV 2.0 GHz	512 MB	Replica
P	Linux kernel 2.6.11	Pentium IV 3.0 GHz	1 GB	Proxy
C	Windows XP Pro SP2	Pentium IV 2.2 GHz	512 MB	Client

We are primarily interested in the request throughput and latency as observed by **C** in five tests. In the first test, each of **L** and **W** sends its response to the client's request to the proxy **P**, which performs output voting on (i.e., compares) these responses before responding to the client. Specifically, in the first test, no system call traces are collected on **L** or **W**, and no behavioral distance is calculated; as such, this serves as a baseline for our tests. In the second test, **L** and **W** additionally capture the system calls made by the web server processes/threads, and send the system call information to another machine (not **P**) for logging and, potentially, offline behavioral-distance calculations. This test thus includes the costs of collecting the system call information and sending it off the server machines, but not the cost of calculating behavioral distances. In the third test, the system call information is sent to proxy **P** (and not to other machines) for online behavioral distance calculation. **P** computes the behavioral distance (in addition to performing output voting, as in the other tests) before responding to the client. In the fourth test, the results of each behavioral distance calculation is cached at **P** so that it need not be performed again if the same system call sequences are received from **L** and **W** in the future. In the last test, only **W** and **C** are used to evaluate the performance of an individual server, in which neither output voting nor behavioral distance is used. We monitor the throughput and latency in each test. The results are shown in Figure 1.

Results from the first test, in which **P** does output voting only, serve as a reference. The second test shows the performance overhead of simply capturing and transporting the system call information off of **L** and **W**. From the results, we can see that this overhead is very small: roughly 1% in throughput and 0.03 millisecond in latency on average. Results of the third test show the overhead of capturing system call information and performing HMM-based behavioral distance calculation on the critical path of responding to the client. As shown, this cost adds substantial overhead to the request processing time. However, the fourth test shows that this cost can be substantially reduced by caching the behavioral distance results. It takes some time for the cache to warm up, and by the end of the test there is less than a 20% throughput loss and 0.59 milliseconds of additional latency on average. Comparing the results of the fourth and the fifth tests, we also see that **L** and **W** are roughly 25% underutilized in the fourth

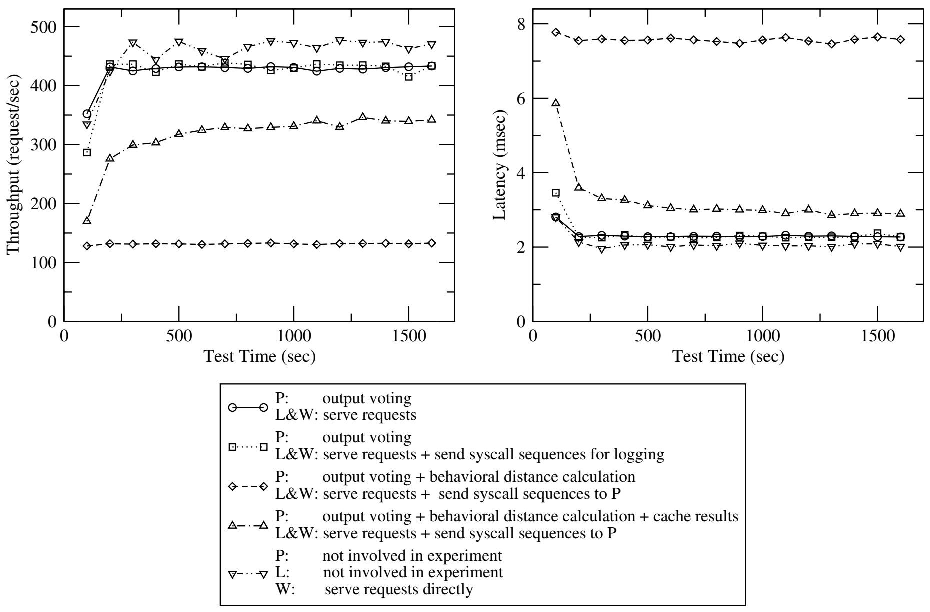

Fig. 1. Performance Overhead of the HMM-based Behavioral Distance

experiment due to the bottleneck created at the proxy. Instantiating the proxy with a faster machine would presumably improve this situation.

6 Conclusion

In this paper we presented a new algorithm for computing behavioral distance between processes. Our approach addresses shortcomings in prior techniques; in particular, it better accounts for system-call orderings while offering comparable performance. Empirical tests suggest that our algorithm offers strong defense against mimicry attacks, while providing substantial improvement in the false-alarm rate over previous proposals. We believe that this algorithm is a significant step toward the practical use of behavioral distance as an anomaly detection technique, particularly for fault- and intrusion-tolerant architectures that already redundantly execute requests on multiple diverse platforms.

References

1. M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 59–74, October 2005.
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. L. E. Baum and T. Petrie. Statistical inference for probabilistic functions of finite state Markov chains. *Ann. Math. Statist.*, 37:1554–1563, 1966.
4. S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.
5. R. W. Buskens and R. P. Bianchini, Jr. 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.
6. C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 2002.
7. M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), November 2002.
8. M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, 21(3), August 2003.
9. L. Chen and A. Avizienis. 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.
10. S. Cho and S. Han. Two sophisticated techniques to improve HMM-based intrusion detection systems. In *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, 2003.
11. B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems – A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, August 2006.
12. R. I. A. Davis, B. C. Lovell, and T. Caelli. Improved estimation of Hidden Markov Model parameters from multiple observation sequences. In *Proceedings of the 16th International Conference on Pattern Recognition (ICPR 2002)*, 2002.
13. 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.
14. 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.
15. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.
16. 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.
17. 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.
18. D. Gao, M. K. Reiter, and D. Song. Behavioral distance for intrusion detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, 2005.
19. J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, 2002.
20. 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.
21. 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.

22. L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.
23. I. M. Meyer and R. Durbin. Comparative ab initio prediction of gene structures using pair HMMs. *Oxford University Press*, 2002.
24. L. Pachter, M. Alexandersson, and S. Cawley. Applications of generalized pair Hidden Markov Models to alignment and gene finding problems. *Computational Biology*, 9(2), 2002.
25. L. R. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. In *Proceedings of IEEE*, February 1989.
26. M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.
27. F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
28. 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.
29. P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26:787–793, 1974.
30. 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.
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. D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.
33. 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.
34. C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.
35. 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.
36. J. Yin, J. 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*, October 2003.

A Updating the b_i Parameters of λ

The idea of updating b_i parameters of λ is the same as of updating a_i (see Section 4.3). Here, we need to calculate the expected number of times λ emits observable symbol $[x, y]$ at q_i, when generating $[S_1, S_2]$. To compute this expectation, we first define a conditional probability, $\zeta([x, y], u, v, i)$, as follows:

$$\zeta([x,y],u,v,i) = \Pr_\lambda\left(\left(\bigvee_{t\geq 0}\begin{array}{l}\mathsf{State}^t = q_i \wedge \\ \mathsf{Out}_1^t = \mathrm{Seq}(x) \wedge \\ \mathsf{Out}_2^t = \mathrm{Seq}(y) \wedge \\ \mathsf{Out}_1^{\leq t} = \mathrm{Pre}(S_1,u) \wedge \\ \mathsf{Out}_2^{\leq t} = \mathrm{Pre}(S_2,v)\end{array}\right) \middle| \begin{pmatrix}\mathsf{Out}_1^{>0} = S_1 \wedge \\ \mathsf{Out}_2^{>0} = S_2\end{pmatrix}\right)$$

where

$$\mathrm{Seq}(x) = \begin{cases}\langle x\rangle & \text{if } x \neq \sigma \\ \langle\rangle & \text{if } x = \sigma\end{cases}$$

and Out_1^t is the sequence of system calls from C_1 in the first component of the emitted symbol in iteration t, with either one (if the component of the emitted symbol is not σ) or zero (if the component of the emitted symbol is σ) system call in the sequence. Out_2^t is defined similarly.

$\zeta([x,y],u,v,i)$ represents the probability of λ being in state q_i after emitting u system calls for process 1 and v system calls for process 2, and the last observable symbol emitted by state q_i is $[x,y]$, given that the system call sequences for process 1 and process 2 are S_1 and S_2, respectively. Note that

$$\gamma(u,v,i) = \sum_{[x,y]} \zeta([x,y],u,v,i)$$

We can calculate $\zeta([x,y],u,v,i)$ easily as follows:

$$\zeta([x,y],u,v,i) = \begin{cases}\frac{\left(\sum_{j=0}^{N}\alpha(u-1,v,j)a_{j,i}b_i([x,\sigma])\right)\beta(u,v,i)}{\Pr_\lambda([S_1,S_2])} & \text{if } x = s_{1,u} \wedge y = \sigma \\ \frac{\left(\sum_{j=0}^{N}\alpha(u,v-1,j)a_{j,i}b_i([\sigma,y])\right)\beta(u,v,i)}{\Pr_\lambda([S_1,S_2])} & \text{if } x = \sigma \wedge y = s_{2,v} \\ \frac{\left(\sum_{j=0}^{N}\alpha(u-1,v-1,j)a_{j,i}b_i([x,y])\right)\beta(u,v,i)}{\Pr_\lambda([S_1,S_2])} & \text{if } x = s_{1,u} \wedge y = s_{2,v} \\ 0 & \text{otherwise}\end{cases}$$

Let the random variable $X_{i,[x,y]}$ be the number of times that state q_i is visited when q_i emits observable symbol $[x,y]$, when λ generates $[S_1,S_2]$. For the same reason as explained in Section 4.3,

$$\mathbb{E}(X_{i,[x,y]}) = \begin{cases}\sum_{u=1}^{l_1}\sum_{v=0}^{l_2}\zeta([x,y],u,v,i) & \text{if } x \neq \sigma \wedge y = \sigma \\ \sum_{u=0}^{l_1}\sum_{v=1}^{l_2}\zeta([x,y],u,v,i) & \text{if } x = \sigma \wedge y \neq \sigma \\ \sum_{u=1}^{l_1}\sum_{v=1}^{l_2}\zeta([x,y],u,v,i) & \text{if } x \neq \sigma \wedge y \neq \sigma\end{cases}$$

and the b_i parameters of λ can be updated as

$$b_i([x,y]) \leftarrow \left(\sum_{k=1}^{M} w_k \mathbb{E}(X_{i,[x,y]}^{(k)})\right) / \left(\sum_{k=1}^{M} w_k \mathbb{E}(X_i^{(k)})\right)$$

B Estimating the Best Mimicry Attack

In this section we show how to estimate the best mimicry attack given an HMM λ. Suppose that the attacker has found a vulnerability in process 2, and wants to use that vulnerability to exploit the process. Let S_2 denote the system call sequence that constitutes the attacker's system calls (e.g., $S_2 = \langle \texttt{open}, \texttt{write} \rangle$). Let $\hat{S}_2$ be an extended sequence of S_2, i.e., $\hat{S}_2$ is obtained by inserting arbitrarily many system calls into S_2 at any locations. When the anomaly detector utilizes HMM-based behavioral distance, a mimicry attack is some $\hat{S}_2$ that induces a large $\Pr_\lambda([S_1, \hat{S}_2])$, where S_1 is the sequence of system calls induced by the attack request at process 1 (not compromised). We assume that S_1 is fixed (vs. being chosen by the attacker), which is typical since for many applications an attack request against process 2 induces an error on process 1 (e.g., a page-not-found error). If the attacker can induce several possible sequences at process 1, then this analysis would need to be repeated with the various alternatives.

For a fixed pair of system call sequences S_1 and $\hat{S}_2$, let $\hat{\Pr}_\lambda([S_1, \hat{S}_2])$ denote the probability of the most probable execution of λ that generates $[S_1, \hat{S}_2]$. Note that $\hat{\Pr}_\lambda([S_1, \hat{S}_2]) < \Pr_\lambda([S_1, \hat{S}_2])$, since multiple executions can yield $[S_1, \hat{S}_2]$ (including that which occurs with probability $\hat{\Pr}_\lambda([S_1, \hat{S}_2])$). Given S_2, there are many different possibilities for $\hat{S}_2$. Each $\hat{S}_2$ has a corresponding $\hat{\Pr}_\lambda([S_1, \hat{S}_2])$. Here we define the "best" mimicry attack, given S_1, S_2 and λ, as the $\hat{S}_2$ that maximizes $\hat{\Pr}_\lambda([S_1, \hat{S}_2])$, i.e., the estimated-best mimicry attack is

$$\arg\max_{\hat{S}_2} \hat{\Pr}_\lambda([S_1, \hat{S}_2])$$

To summarize, in order to find the estimated-best mimicry attack, we need to try different possible $\hat{S}_2$ sequences, and different executions of the HMM in generating $[S_1, \hat{S}_2]$ in order to find the one that results in the highest probability. Here we propose an efficient algorithm to do this.

We first try to find the estimated-best $\hat{S}_2$, by considering ways to improve a given mimicry attack, i.e., to modify $\hat{S}_2$ to increase $\hat{\Pr}_\lambda([S_1, \hat{S}_2])$. This can be achieved by changing the way a transition is made from any state q_i to q_j when generating $[S_1, \hat{S}_2]$. Since we are modifying an existing mimicry attack, we want to make sure that the modification does not emit any system calls in S_1, otherwise the mimicry attack will fail (though the modification can emit additional system calls for process 2).

There are basically two ways to transition from q_i to q_j: an execution of the HMM makes a transition from q_i to q_j directly with probability $a_{i,j}$; or an execution makes a transition from q_i to q_j indirectly by visiting some states in the HMM (and emitting some observable symbols). Note that in the latter case, the observable symbols emitted for process 1 need to be σ's, while the symbols emitted for process 2 can be any system calls in C_2. In order to find the best way (the one with highest probability), we define

$$\hat{a}_{i,j}(e) = \max\left(\left\{\Pr_\lambda\left(\bigvee_{t_2>t_1\geq 0}\begin{array}{l}\mathsf{State}^{t_1} = q_i \ \wedge\\ \mathsf{State}^{t_2} = q_j \ \wedge\\ \mathsf{Out}_1^{>t_1\wedge<t_2} = \langle\rangle \ \wedge\\ \mathsf{Out}_2^{>t_1\wedge<t_2} = S\end{array}\right)\right\}_{S\neq\langle\rangle \ \wedge \ e\notin S} \cup \{a_{i,j}\}\right)$$

where $\langle\rangle$ represents an empty sequence, and S is any non-empty sequence of system calls from $(C_2 \setminus \{e\})$. $\mathsf{Out}_1^{>t_1\wedge<t_2}$ is the sequence of system calls from C_1 in the first components of the emitted symbols (less σ) between iteration t_1+1 and iteration t_2-1, and similarly for $\mathsf{Out}_2^{>t_1\wedge<t_2}$. $\hat{a}_{i,j}(e)$ represents the highest probability of emitting any system calls for process 2 except e, while emitting no system call (only a sequence of σ) for process 1, when transitioning from q_i to q_j. (It may not be clear now why a special system call e needs to be excluded. We will explain this later in this section.) Note that a special case is when S is empty, which corresponds to transitioning from q_i to q_j directly.

$\hat{a}_{i,j}(e)$ can be solved efficiently by solving for all-pairs shortest paths in a graph $G = \langle V, E\rangle$, where V consists of two nodes q_i^{in} and q_i^{out} for every state q_i in the HMM, and the cost $c(n_1, n_2)$ for each edge (n_1, n_2) is defined as

$$c(n_1,n_2) = \begin{cases} |\log a_{i,j}| & \text{if } n_1 = q_i^{\mathsf{out}} \wedge n_2 = q_j^{\mathsf{in}} \\ |\log \hat{b}_i(\sigma, e)| & \text{if } n_1 = q_i^{\mathsf{in}} \wedge n_2 = q_i^{\mathsf{out}} \\ \infty & \text{otherwise} \end{cases}$$

where

$$\hat{b}_i(x,e) = \max_{c\in(C_2\cup\{\sigma\}\setminus\{e\})} b_i([x,c])$$

That is, $\hat{b}_i(x,e)$ is the highest probability of emitting x from process 1 and any system call (including σ and excluding e) from process 2 at state q_i.

With $\{\hat{a}_{i,j}(e)\}$ calculated, the algorithm of finding the estimated-best mimicry attack becomes very similar to the algorithm of finding $\Pr_\lambda([S_1, S_2])$ (see Section 4.2). The differences are

– In computing $\Pr_\lambda([S_1, S_2])$ we only allow σ to be inserted into S_1 and S_2, but here we allow σ and any system calls to be inserted into S_2 (for S_1 it remains the same — only σ is allowed).
– In computing $\Pr_\lambda([S_1, S_2])$ we consider all executions of the HMM, and sum up the corresponding probabilities. Here we consider only one execution that generates S_1 and S_2 with the highest probability.

We define $\delta(u, v, i)$ to be the probability of the most probable mimicry execution to generate exactly the first u system calls of S_1, and exactly the first v system calls of S_2, when the current state is q_i, among all executions. As a technical matter, when computing $\delta(u, v, i)$ inductively, we need to take care to ensure that the HMM executions considered in the calculation of $\delta(u, v, i)$ do not include those that should be considered only in calculating $\delta(u, v', i)$ for

$v' > v$. Intuitively, the danger is HMM executions that, in the course of emitting arbitrary system calls before reaching the next attack system call in S_2, in fact insert attack system calls from S_2 as these "arbitrary" system calls. It is for this reason that in calculating $\delta(u, v, i)$ inductively, we need to exclude HMM executions that output elements of S_2 prematurely, hence the arguments to $\hat{a}_{i,j}$ and $\hat{b}_i$. Given this, $\delta(u, v, i)$ can be solved inductively as follows.

Base cases:

$$\delta(0,0,i) = \begin{cases} 1 & \text{if } i = 0 \\ 0 & \text{otherwise} \end{cases} \qquad \delta(u,v,0) = \begin{cases} 1 & \text{if } u = v = 0 \\ 0 & \text{otherwise} \end{cases}$$

Induction:

$$\delta(u,0,i) = \max_{j\in[0,N]} \left(\left\{ \delta(u-1,0,j)\hat{a}_{j,i}(s_{2,1})\hat{b}_i(s_{1,u},s_{2,1}) \right\} \right) \qquad \text{for } \begin{matrix} u > 0, \\ i > 0 \end{matrix}$$

$$\delta(0,v,i) = \max_{j\in[0,N]} \left(\left\{ \delta(0,v-1,j)\hat{a}_{j,i}(s_{2,v})b_i([\sigma,s_{2,v}]) \right\} \right) \qquad \text{for } \begin{matrix} v > 0, \\ i > 0 \end{matrix}$$

$$\delta(u,v,i) = \max_{j\in[0,N]} \left(\begin{matrix} \left\{ \delta(u-1,v,j)\hat{a}_{j,i}(s_{2,v+1})\hat{b}_i(s_{1,u},s_{2,v+1}) \right\} \cup \\ \{ \delta(u,v-1,j)\hat{a}_{j,i}(s_{2,v})b_i([\sigma,s_{2,v}]) \} \cup \\ \{ \delta(u-1,v-1,j)\hat{a}_{j,i}(s_{2,v})b_i([s_{1,u},s_{2,v}]) \} \end{matrix} \right) \qquad \text{for } \begin{matrix} u,v > 0, \\ v < l_2, \\ i > 0 \end{matrix}$$

$$\delta(u,v,i) = \max_{j\in[0,N]} \left(\begin{matrix} \left\{ \delta(u-1,v,j)\hat{a}_{j,i}(\perp)\hat{b}_i(s_{1,u},\perp) \right\} \cup \\ \{ \delta(u,v-1,j)\hat{a}_{j,i}(s_{2,v})b_i([\sigma,s_{2,v}]) \} \cup \\ \{ \delta(u-1,v-1,j)\hat{a}_{j,i}(s_{2,v})b_i([s_{1,u},s_{2,v}]) \} \end{matrix} \right) \qquad \text{for } \begin{matrix} u > 0, \\ v = l_2, \\ i > 0 \end{matrix}$$

Then, $\hat{\Pr}_\lambda([S_1, \hat{S}_2])$ of the estimated-best mimicry attack given S_1, S_2 and λ is

$$\max_{i\in[1,N]} \left(\{ \delta(l_1,l_2,i)\hat{a}_{i,N+1}(\perp) \} \right)$$

The above inductive algorithm is efficient in calculating $\hat{\Pr}_\lambda([S_1, \hat{S}_2])$. Moreover, by recording the most probable $\hat{S}_2$ (i.e., prefix of the eventual, estimated-best mimicry) for each step of the induction, we can efficiently obtain the estimated-best mimicry attack in the sense we have described.

An interesting question is whether this algorithm can be extended to find the "real" best mimicry attack. To do so, the corresponding $\delta'(u, v, i)$ needs to be defined as the "highest sum of probabilities of all executions" for (u, v, i). However, in assembling the most probable mimicry as discussed above, do we record $\delta'(u, v, i)$ for one particular $\hat{S}_2$, or $\delta'(u, v, i)$ for all possible $\hat{S}_2$'s? Unfortunately, the latter is required, because when calculating $\delta'()$ of larger indices, we need the results of $\delta'()$ of lower indices for different $\hat{S}_2$'s. Since for each (u, v, i) we need to record $\delta'(u, v, i)$ for all possible $\hat{S}_2$'s, this algorithm requires exponential computation time and memory in the worst case in the length of the best mimicry. As such, we presently settle for the "estimated-best" mimicry attack, which showed how to compute efficiently above, and leave finding the absolute best mimicry attack to future work.