Beyond Output Voting: Detecting Compromised Replicas Using HMM-Based Behavioral Distance

Debin Gao, Michael K. Reiter, *Senior Member*, *IEEE Computer Society*, and Dawn Song

Abstract—Many host-based anomaly detection techniques have been proposed to detect code-injection attacks on servers. The vast majority, however, are susceptible to "mimicry" attacks in which the injected code masquerades as the original server software, including returning the correct service responses, while conducting its attack. "Behavioral distance," by which two diverse replicas processing the same inputs are continually monitored to detect divergence in their low-level (system-call) behaviors and hence potentially the compromise of one of them, has been proposed for detecting mimicry attacks. In this paper, we present a novel approach to behavioral distance measurement using a new type of Hidden Markov Model, and present an architecture realizing this new approach. We evaluate the detection capability of this approach using synthetic workloads and recorded workloads of production web and game servers, and show that it detects intrusions with substantially greater accuracy than a prior proposal on measuring behavioral distance. We also detail the design and implementation of a new architecture, which takes advantage of virtualization to measure behavioral distance. We apply our architecture to implement intrusion-tolerant web and game servers, and through trace-driven simulations demonstrate that it experiences moderate performance costs even when thresholds are set to detect stealthy mimicry attacks.

Index Terms—Intrusion detection, replicated system, output voting, system call, behavioral distance.

1 Introduction

Many host-based anomaly detection systems have been proposed to detect server compromises, e.g., code injection attacks exploiting buffer overflow or format-string vulnerabilities [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. These systems detect intrusions by monitoring the execution of a program to see if its behavior conforms to a model that describes its normal behavior. Constructing such a model for accurate intrusion detection is challenging, especially due to *mimicry attacks* [11], [12], [13], [14]. In a mimicry attack, the injected attack code masquerades as the original server software, including returning the correct service responses, so that the anomaly detector cannot differentiate execution of the attack code from execution of the original server program. Output voting in a replicated system that detects [15], [16], [17] or masks [18], [19], [20], [21], [22], [23], [24] intrusions by comparing server outputs cannot detect such attacks either. A replicated system that employs only output voting will thus allow a compromised server that generates the correct output to conduct other types of attacks, e.g., to attack other machines in the local network.

Behavioral distance [25], [26] has been proposed to detect carefully crafted mimicry attacks that would evade detection by a system that utilizes traditional host-based anomaly detection or output voting. This approach compares the low-level behaviors (e.g., system calls) of two diverse replicas when processing the same, potentially malicious, inputs. If the two replicas are diverse and vulnerable only to different exploits, a successful attack on one of them might induce a detectable increase in the behavioral distance. This makes mimicry attacks potentially more difficult, because to avoid detection, the behavior of the compromised process must be close to the behavior of the uncompromised one. The initial proposal for behavioral distance [25] is based on measuring the evolutionary distance (ED) [27] between replicas' observable behaviors.

In this paper, we present an alternative approach based on a novel Hidden Markov Model (HMM) for computing behavioral distance, and present the design, implementation, and evaluation of a novel architecture using HMM-based behavioral distance to detect attacks. 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).

An interesting and important observation is that since these hidden tasks should be the same (if the processes are

- *D. Gao is with the School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore 178902. E-mail: dbgao@smu.edu.sg.*
- *M.K. Reiter is with the Department of Computer Science, University of North Carolina at Chapel Hill, Campus Box 3175, Sitterson Hall, Chapel Hill, NC 27599-3175. E-mail: reiter@cs.unc.edu.*
- *D. Song is with the Computer Science Division, University of California, Berkeley, 675 Soda Hall, Berkeley, CA 94720-1776. E-mail: dawnsong@cs.berkeley.edu.*

Manuscript received 8 Feb. 2008; revised 23 June 2008; accepted 8 July 2008; published online 21 July 2008.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-2008-02-0029. Digital Object Identifier no. 10.1109/TDSC.2008.39.

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., [28] and [29]), where an HMM models a single process.

We detail the behavioral 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 attack against our HMM, and evaluate the false-alarm rate when the behavioral-distance threshold is set to detect this estimated-best mimicry. In order to make a fair comparison between our new approach and previous approaches, we first evaluate detection capability of the HMM approach using synthetic web workloads. We show that our new approach yields better results than the ED approach [25], in many cases offering substantial improvement in the false-alarm rate. We additionally evaluate the false-alarm rate of the system using recorded workloads of production web and game servers. We show, for example, that a web server using the HMM-based behavioral distance, when configured to detect the "best" mimicry attacks, yielded as few as three false alarms when processing a recorded workload of over two million client requests. Similarly configured, a game server yielded 14 false alarms when processing 39,000 recorded game events. As such, we argue that the HMM approach offers substantially superior properties for calculating behavioral distance for anomaly detection.

We also present the design, implementation, and evaluation of a novel architecture to detect mimicry attacks using behavioral distance. We address the systems issues necessary to make this technique practical by presenting a complete architecture based on virtualization for monitoring the system-call behaviors of diverse replicas, and for efficiently evaluating their behavioral distance either on or off the critical path of responding to clients. In particular, we detail the various components of the architecture, how they communicate, and the responsibilities of each.

We demonstrate our architecture through the implementation and evaluation of a web server and a game server. These servers present distinct challenges in many ways. For example, the web server is a typical request-response server, making it convenient to compute the distance between replicas' behaviors when processing the same request. In contrast, the game server's responses are not in one-to-one correspondence with client requests, which makes it necessary to pair the low-level behaviors of replicas via alternative means for computing their behavioral distances. The typical workload and performance requirements for these servers are also quite different, e.g., a typical web server generates relatively long responses of a few kilobytes to a few hundred kilobytes, and throughput is critical as it may need to provide service to a large number of users simultaneously. In contrast, the game server generates much shorter responses of less than a hundred bytes long, and is required to do so primarily with a short latency. Consequently, our evaluation sheds light on the suitability of our architecture for two very different types of servers.

In an evaluation of performance overhead of our architecture, our web server's throughput drops to about 50 percent compared to a standalone web server on the same physical machine without any protection mechanism, and players experience an overhead of 8 to 86 milliseconds (ms) in additional latency for our game server with 128 to 1,024 concurrent players.

In summary, this paper makes the following contribution:

- We present a novel approach for measuring behavioral distance using a new type of HMM.
- With the evaluation using synthetic web workloads and recorded traces of production web and game servers, we show that this new approach based on HMM detects intrusions with substantially greater accuracy than previous approaches [25].
- We demonstrate the design and implementation of a novel architecture employing behavioral distance using a web server and a game server.
- We show that the proposed architecture using behavioral distance is practical for real servers and able to detect compromised servers with moderate overhead.

2 Related Work

N-variant systems [30] are closely related to our work. An N-variant system executes a set of automatically diversified variants on the same inputs, and monitors their behavior to detect divergence. By constructing variants so that an anticipated type of exploit can succeed on only one variant, the exploit can be rendered detectable. The construction of these variants usually requires a special compiler or a binary rewriter, but perhaps more importantly, it detects only anticipated types of exploits, against which the replicas are diversified. Similarly, Cavallaro [31] proposes monitoring a process and a diversification of the process to detect behavioral divergences triggered by memory error exploits, to provide deterministic protection. The system we propose here, instead, uses behavioral distance to detect potentially unforeseen types of compromises of one of two off-the-shelf servers.

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.

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

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 [32]. HMMs have also been used in intrusion detection systems, e.g., to model the system-call behavior of a single process [28], [33], and to model privilege flows [29]. 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" [34] and "generalized pair HMMs" [35] 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.

Numerous systems have employed output voting to detect some types of server compromises. For example, the HACQIT system [36], [37] uses two web servers, the Microsoft Internet Information Server (IIS) and the Apache web server, to detect, isolate, and possibly recover from software failures. If the status codes of the replica responses are different, the system detects a failure. This idea was extended by Totel et al. to do a more detailed comparison of the replica responses [38]. They realized that web server responses may be slightly different even when there is no attack, and proposed a detection algorithm to detect intrusions with a higher accuracy. These projects specifically target web servers and analyze only server responses. Consequently, they cannot detect a compromised replica that responds to client requests consistently, while attacking the system in other ways. Our system, in contrast, monitors low-level behaviors (system calls) of the replicas, and is applicable to virtually any services (not just web servers).

There are also a long list of techniques proposed for host-based intrusion detection by dynamically monitoring system calls. Some rely on control-flow information by monitoring the sequence of system calls made [1], [2], [4], [6], [7], [10], and other use data-flow information by monitoring system-call arguments [39], [40], [41], [42], [43]. Our work is different from these approaches that we introduce diversity into a replicated system for intrusion detection.

3 Behavioral Distance Measurement Using a Hidden Markov Model

In this section, we first explain the motivation for this new approach using HMMs for measuring behavioral distance. After that, we will present the new HMM and its use in measuring behavioral distance in detail. In the end, we will discuss some implementation issues.

3.1 Motivation for Our Approach

To motivate our approach, we first briefly describe the problem we are trying to solve. In a nutshell, the problem is to assign a distance to a pair of system-call sequences

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

emitted by two processes while processing the same input. Here, each $s_{i,j}$ denotes the system-call number (a natural number) of the jth system call by the ith process. The distance should indicate whether these sequences reflect similar activities. Note that we make use of only the system-call numbers and not the system-call arguments and return values, for simplicity. Consequently, only attacks modifying the control flow in one replica, or more specifically, changing its system-call behavior, can be detected. Attacks exploiting system-call arguments, persistent interposition attacks [44], and non-control-data attacks [45] will not change this distance. Though restricting our attention to only system-call numbers does simplify our task, producing the distance between two system-call sequences 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 [25] 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*:

$$\left\langle s'_{1,1}, s'_{1,2}, \ldots, s'_{1,l'_1} \right\rangle \qquad \left\langle s'_{2,1}, s'_{2,2}, \ldots, s'_{2,l'_2} \right\rangle, \quad (2)$$

where $l'_1 \ge l_1$, $l'_2 \ge l_2$, and $l'_1 = l'_2$. The distance for alignment (2) was simply $\sum_j \mathrm{dist}(s'_{1,j}, s'_{2,j})$, where $\mathrm{dist}(\cdot)$ 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 ED. Since system-call order is known to be important to detecting intrusions (e.g., [1], [4], [7], and [10]), 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 $\mathrm{dist}(\cdot)$ table in the ED approach. However, in HMM-based behavioral distance, the probability of emitting the same observable symbol is generally different for different hidden states, whereas in ED-based behavioral distance, a universal $\mathrm{dist}(\cdot)$ 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 the use of HMM 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 3.2.

3.2 The Hidden Markov Model

In this section, we introduce our HMM and describe how it is used for behavioral distance calculation. We begin in Section 3.2.1 with an overview of the HMM. We then present our algorithm for calculating the behavioral distance in Section 3.2.2, and describe the original construction of the HMM in Section 3.2.3.

3.2.1 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 $A = \{a_i\}_{0 \le i \le N}$ of state transition probability distributions. Each $a_i : \{1, \ldots, N+1\} \to [0, 1.0]$ satisfies $\sum_j a_i(j) = 1.0$. $a_i(j)$ 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 $B = \{b_i\}_{1 \le i \le N}$ of symbol emission probability distributions. Each $b_i : (C_1 \cup \{\sigma\}) \times (C_2 \cup \{\sigma\}) \to [0, 1.0]$ satisfies $\sum_{[x,y]} b_i([x, y]) = 1.0$. $b_i([x, y])$ 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 3.1, 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 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 an HMM λ, there are many executions that generate the given pair of sequences $[S_1, S_2]$ as in (1). We use $\mathrm{Ex}_\lambda([S_1, S_2])$ to denote the set of executions of λ that generate $[S_1, S_2]$. The probability that λ generates the sequences $[S_1, S_2]$ (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 (1). Note that

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

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

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

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 $t + 1$).

3.2.2 Computing $\Pr_\lambda([S_1, S_2])$

$\Pr_\lambda([S_1, S_2])$ is the probability that system-call sequences S_1 and S_2 are generated (in the sense of Section 3.2.1) by the HMM λ, which is used as the behavioral distance between S_1 and S_2.[5] 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 using dynamic programming, given λ, S_1, and S_2. How we build λ itself is the topic of Section 3.2.3.

4. In Section 3.3, 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.

5. Note that although we call this *behavioral distance*, it generally does not have the mathematical properties of a *distance*. For example, behavioral distance has the positivity property (all distances are nonnegative), but does not obey symmetry or the triangle inequality.

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 nonzero 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 nonzero probability. Moreover, even for one particular state sequence, there are many ways of generating S_1 and S_2 with σ being 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 $\mathrm{Out}_1^{\leq t}$ 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 $\mathrm{Out}_1^{\leq t}$ 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 $\mathrm{Out}_2^{\leq t}$ would be the non-σ values in $\langle s'_{2,1},\ldots,s'_{2,\ell}\rangle$. Now define

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

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:

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

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

Induction:

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

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

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

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 $j\in\{1,\ldots,N\}$, 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 $v-1$ 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 $b_i([s_{1,u},s_{2,v}])$).
- 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 $b_i([\sigma,s_{2,v}])$).

After $\alpha(u,v,i)$ is solved for all values of $u\in\{0,1,\ldots,l_1\}$, $v\in\{0,1,\ldots,l_2\}$, and $i\in\{1,\ldots,N\}$, 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

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

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 $\mathrm{Out}_1^{>t}$ 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 $\mathrm{Out}_2^{>t}$. 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.

3.2.3 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 Supplemental Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2008.39.

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, 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 $i \le N$, $j \le N + 1$, as follows:

$$\gamma(u,v,i) = \Pr_\lambda\left(\begin{pmatrix}\mathrm{State}^t = q_i \wedge \\ \bigvee_{t\ge 0} \mathrm{Out}_1^{\le t} = \mathrm{Pre}(S_1,u) \wedge \\ \mathrm{Out}_2^{\le t} = \mathrm{Pre}(S_2,v)\end{pmatrix} \middle| \begin{pmatrix}\mathrm{Out}_1^{>0} = S_1 \wedge \\ \mathrm{Out}_2^{>0} = S_2\end{pmatrix}\right)$$

$$\xi(u,v,i,j) = \Pr_\lambda\left(\begin{pmatrix}\mathrm{State}^t = q_i \wedge \\ \mathrm{State}^{t+1} = q_j \wedge \\ \bigvee_{t\ge 0} \mathrm{Out}_1^{\le t} = \mathrm{Pre}(S_1,u) \wedge \\ \mathrm{Out}_2^{\le t} = \mathrm{Pre}(S_2,v)\end{pmatrix} \middle| \begin{pmatrix}\mathrm{Out}_1^{>0} = S_1 \wedge \\ \mathrm{Out}_2^{>0} = S_2\end{pmatrix}\right).$$

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 3.2.2, 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:

$$\gamma(u,v,i) = \frac{\alpha(u,v,i)\beta(u,v,i)}{\Pr_\lambda([S_1,S_2])}$$

$$\xi(u,v,i,j) = \frac{\alpha(u,v,i)a_{i,j}}{\Pr_\lambda([S_1,S_2])} \times \begin{pmatrix} b_j\big([s_{1,u+1},\sigma]\big)\beta(u+1,v,j) + \\ b_j\big([\sigma,s_{2,v+1}]\big)\beta(u,v+1,j) + \\ b_j\big([s_{1,u+1},s_{2,v+1}]\big)\beta(u+1,v+1,j)\end{pmatrix}.$$

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, $\mathbb{E}(X_i^{u,v}) = \sum_{x\in\{0,1\}} x \Pr(X_i^{u,v} = x) = \gamma(u, v, i)$. Then, by linearity of expectation

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

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 [46], as follows:

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

This equation shows 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 $([S_1^{(1)}, S_2^{(1)}], \ldots, [S_1^{(M)}, S_2^{(M)}])$, we can calculate the relevant parameters for each pair of sequences (i.e., $\mathbb{E}(X_i^{(k)})$, $\mathbb{E}(X_{i,j}^{(k)})$) and then update the a_i parameters of λ as

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

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 [47]. 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

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

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 a_i 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 4.1).

3.3 Implementation Issues

There are several implementation issues that deserve comment. First, in all discussions 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 [2], [7], [25]. In our experiments, we use the same phrase-extraction algorithm as in the ED project [25]. 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 4 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 3.2.3, 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.

4 Detection Capability of the HMM-Based Behavioral Distance Measurement

As discussed in Section 3, 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 [25]. In this section, we evaluate an implementation of our anomaly detector using HMM-based behavioral distance to determine whether this is, in fact, true. In order to make a fair comparison between this evaluation and previously reported results for the ED-based approach [25], we first use synthetic workloads generated by a benchmarking tool (the same workload used in previous work). After that, we perform another set of evaluations using real workload to see how practical our system is in protecting real web and game servers.

Our evaluation system includes two computers running web or game servers to process client requests. One of these computers, denoted **L**, runs Linux and the other, denoted **W**, runs Windows. Each of **L** and **W** was given the same sequence of requests and each recorded the system-call sequence, denoted by $S_{\mathbf{L}}$ and $S_{\mathbf{W}}$,[6] respectively, of (the thread in) the 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 3.2.3.

4.1 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 [25], 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()` system call. Note that we have considered only this attack sequence in this evaluation in order to compare our results to the ED-based project, and leave the evaluation of other attack sequences in future work.

To measure the false-alarm rate when detecting the best mimicry attack, 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 largest 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.

6. System calls on Windows are also called native API calls or kernel calls. Section 5.1.1 explains how the Windows system-call information is obtained.

TABLE 1
False-Alarm Rate when Detecting the Estimated-Best Mimicry

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 %

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 Supplemental Appendix B, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TDSC.2008.39. 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 [25] as well). An attack for which the attack activity spans multiple requests or multiple server processes/threads is an area of ongoing work.

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. Since the system detects the "best" mimicry, it experiences a zero false-negative rate. 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.

4.2 Evaluation Using Synthetic Workloads

Our first evaluation uses synthetic workloads generated from the static test suite of WebBench 5.0,[7] which is exactly the same workloads used by the evaluation of the ED approach [25]. 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.) Behavioral distance is measured on the system-call sequences resulted from the processing of HTTP requests. Table 1 presents the 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.

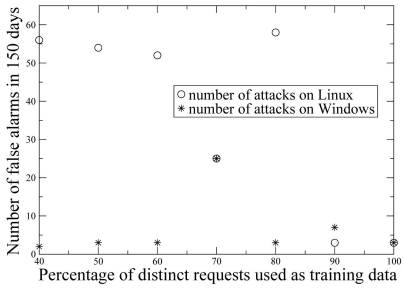


Fig. 1. Number of false alarms when detecting the "best" mimicry attack.

4.3 Evaluation Using Recorded Traces of Production Web and Game Servers

Although the evaluation using synthetic workloads presented in the previous section gives a good idea about intrusion detection capability of our approach, it remains to evaluate the system using real traces. In this part of the evaluation, we use recorded traces of production web and game servers to evaluate the detection capability of the HMM-based approach of measuring behavioral distance, and show that the HMM approach results in very low false-alarm rate when used on production web and game servers.

4.3.1 Web Server

The recorded trace we use consists of a five-month-long log of client requests for static pages on www.cylab.cmu.edu. This data consists of more than two million requests on about 2,700 distinct URLs, including `html` pages, images, videos, etc. In this test, we use a training set to build the HMM, a validation set to detect overfitting the training data, and a testing set to evaluate the accuracy of the model. The training set contains a subset (of a size that varies per experiment; see next paragraph) of the 2,700 distinct URLs. We request each URL in this subset once, and use the system-call sequences induced to train the HMM. The validation set consists of all URLs on a typical weekday, which has about 12,000 requests. After the model is built using the training set and the validation set, it is evaluated on the testing set which is simply the entire trace data set excluding the validation set. Similar to the evaluation using synthetic workloads, behavioral distance is measured on the system-call sequences resulted from the processing of HTTP requests. Both replicas run Apache `httpd` 2.2.2.[8]

We perform the test using varying percentages of the possible distinct requests as training data, to simulate the scenario in which new contents are added to a web server but the behavioral distance model is not retrained (and so the training set does not contain all the distinct requests). Fig. 1 shows the number of false alarms when the training set consists of 40 percent to 100 percent of the distinct

7. VeriTest, http://www.veritest.com/benchmarks/webbench/default.asp.

8. Apache on Linux and Apache on Windows are different code bases.

requests, when the system is tested on about two million requests recorded in 150 days.

These results suggest that our system detects software intrusions with high accuracy. Our system generates only three false alarms in more than two million requests, when the training set consists of all distinct requests. When 20 percent of the requests are not included in the training set, the number of false alarms increase to about 60, which is still very good. These results are also about an order of magnitude better than those previously reported [25]. They suggest that the model be retrained when the training set consists of less than 90 percent of the distinct requests, if very low false-alarm rates are required.

4.3.2 Game Server

A web server is one of the most common services provided over the Internet, and therefore is a typical example in which behavioral distance is useful for defending against software intrusions. However, it is also relatively simple in that each transaction consists of a single request and a response. In this part of the evaluation, we show another application in which behavioral distance is used to protect an online multiplayer game server. This is more complicated because a message from a player may result in zero or multiple responses to the sender as well as other players. Dynamic generation of server responses also makes it more complex.

The online game server we choose to work with is the Peekaboom game server (www.peekaboom.org). Peekaboom [48] is an online game for two players (single-player games are also possible; please see www.peekaboom.org for details), in which one of the players (Boom) continuously reveals parts of an image, and the other player (Peek) tries to guess the word that is associated with the image. The Peekaboom server is implemented in Java, and so is theoretically immune to the code injection attacks that are a primary motivation for our work. However, Peekaboom is the only server available to us that is both representative of more complex, dynamic services and accessible for recording traces. We believe that both the adaptation of our architecture to this application and its evaluation (Section 6.2) provide a realistic view of the suitability of our approach to similar services written in C/C++, for example.

The recorded games describe the actions players performed in a game. We developed an automatic player program to replay these recorded games to generate requests to the system. For each new game, the game server chooses an image and a label for the chosen image. Our automatic player program then searches the recorded games to locate those for the given image and label, and then chooses one of the games and replays the client requests. Due to the complexity of the game server, behavioral distance is measured on the system calls resulted from the processing of *game events* instead of the processing of client requests. Please see Section 5.2.1 for more discussion.

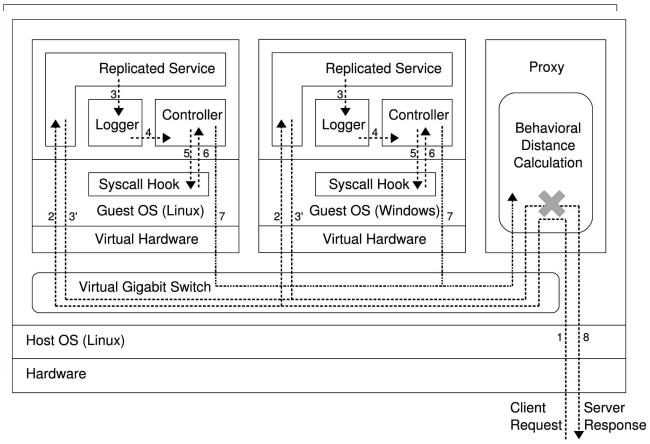
Our detection accuracy tests for the Peekaboom server are similar to those for Apache. We obtain the original source code of Peekaboom to implement two replicas running the Windows and Linux operating systems, respectively. We (randomly select and) replay the recorded games and collect system-call information when the replicated servers process each game event. We collected system-call sequences for over 60,000 game events on each replica, out of which about 10,000 were used for training, about 11,000 were used for validating, and the remaining 39,000 were used for testing. That is, an HMM is built using the training and validation sets; the threshold of the system is set to detect the "best" mimicry attack; and the model is then evaluated on the testing set. During testing, we recorded 14 false alarms; the same number of false alarms is recorded for mimicry attacks on Linux and Windows.

These results were obtained when we use a single HMM to model the server behaviors in all game events. Our examination of the system, however, revealed a potentially more effective approach for the game server, namely one using a distinct model per game event type. There are 19 different types of game events. One such event type is a request parsing event that is invoked when the game server receives a client request. During this event, the game server preprocesses the request to create a game event object that describes the request, and then passes it to the corresponding event handler. We expect this request parsing event to be the only event that occurs on the uncompromised replica when an attack message is received, since for the types of attacks we anticipate, the attack invocation should be treated as malformed by the uncompromised replica. In this case, the attack system calls must be made during the processing of the request parsing event on the uncompromised replica; otherwise, behavioral distance will detect an anomaly since only the request parsing event is observed on the uncompromised replica. Moreover, neither replica makes a `write()` system call during the request parsing event. As such, the attack we consider (in which the attacker attempts an `open()` followed by a `write()`) would always be detected if performed during the attack invocation, if the proxy checks that the two replicas perform the same types of game events and maintains the set of system calls that are allowed during processing each event type on each replica. Moreover, if this set for each event type is complete, this model should yield *no* false alarms.

This alternative behavioral distance calculation is possible because we are able to obtain fine-grained event type information from the Peekaboom game server on both replicas. This would be nontrivial if the replicas were running different code bases. Another limitation of our analysis is that we have considered only one type of mimicry attack, albeit one (`open()`, `write()`) that is seemingly the least an attacker must do to modify or create data on the system being protected.

5 System Architecture for Behavioral Distance

In Sections 3 and 4, we present the HMM approach and an evaluation of the approach on its detection capability and performance overhead. In this section, we focus on the systems issues and introduce a novel architecture for behavioral distance. A system utilizing behavioral distance consists of at least two replicas and a proxy. The replicas run servers either on different operating systems or with



Msg 1: request from a client
Msg 2: duplicated request from the proxy
Msg 3: log from a replicated server
Msg 3': response from a replicated server
Msg 4: log from the logger
Msg 5: request for syscall info
Msg 6: system call info from the kernel
Msg 7: syscall info from the controller
Msg 8: response from the proxy

Fig. 2. Architecture of the system.

different programs. The proxy is a gateway between the replicas and clients.

Our architecture hosts the replicas and the proxy on a single physical machine, using virtualization. Doing so can minimize the network delays between the replicas and the proxy, because these delays are limited only by the speed of memory copies with virtualization. Since there are many messages exchanged between each replica and the proxy, the savings can be significant. Other benefits include better resource management among the proxy and the replicas (this resource sharing is handled by the scheduler on the host operating system automatically) and reduction of hardware and maintenance costs. Although using virtualization could potentially open up another type of attacks, e.g., attacks against the virtualization platform, here we only focus on protecting the application servers running on the virtual machines (e.g., web servers and game servers).

We configure two virtual machines, one for each replica, and run the proxy directly on the host (running Linux). Fig. 2 shows the system architecture and the messages involved in a client request/response. Upon receiving a request (Msg 1), the proxy forwards it (Msg 2) to both replicas. A replicated server processes the request and sends a response (Msg 3') to the proxy. At the same time, the server sends a log (Msg 3) containing information about the request to the logger, which forwards it (Msg 4) to the controller. The controller processes the log, requests (Msg 5) and receives (Msg 6) system-call information for the corresponding request, and forwards the system-call information (Msg 7) to the proxy. The proxy does output voting on the server responses and behavioral distance measurement on the system-call sequences. If either fails, i.e., if either the responses are different or the behavioral distance is greater than a predefined threshold, the response will be blocked and an alarm will be set off; otherwise, the response (Msg 8) will be sent to the client. The proxy also maintains a cache that remembers the behavioral distance calculation results.

5.1 Web Server Implementation

In this section, we detail how we have applied this architecture to protect Apache web servers. The two replicas in this system run Apache `httpd` on Linux (multiprocess) and Windows (multithreaded), respectively. A process/thread is assigned to handle each request. Our system measures the behavioral distance between the system calls of the process and thread that serve the same request.

5.1.1 System-Call Hook

To capture system calls on Linux, we modify the kernel source to record system calls made by a program and save the system-call numbers in the kernel space. A new system call[9] is used for a user program running as `root` (the controller, see Section 5.1.3) to send commands to the kernel to start/stop system-call interception and to request system-call information recorded for a process ID. Upon receiving a request, the kernel sends all system-call numbers recorded for the process ID to the user program via a UNIX pipe.

On Windows, system call[10] hooking is implemented as a kernel driver, which overwrites the KiSystemService table. The KiSystemService table contains the addresses of all system-call-handling functions. System-call information is extracted by overwriting them with the addresses of new system-call-handling functions, which simply save the system-call numbers in the kernel and then invoke the original handling functions. Unlike Linux, Windows provides an interface for a user program to send requests to and receive responses from a kernel driver. Thus, we do not have to implement a new system call to do this.

5.1.2 Logger

One of the most difficult tasks in implementing such a system for real-time behavioral distance measurement is to match a system-call sequence with its corresponding `http` request. This is nontrivial because there could be many requests being processed simultaneously by different processes (or threads); therefore, simply using the timing information would not reliably match the system-call sequences with their corresponding requests. Instead, we insert a tag into each request when it first enters the system and trace the tag to match system-call sequences with their corresponding requests.

The tag, which is just a unique index number, is inserted into the `http` header by the proxy. Since a proxy has to insert its proxy information anyway according to the `http` RFC, this does not add much overhead. After inserting the tag, we let Apache log the value of the tag and the process ID of the process (or the thread ID of the thread) that served the request and send this information to the logger.[11] Upon receiving the tag and the process/

9. We utilize a system-call number that is reserved and not implemented yet on the 2.6.15 Linux kernel.

10. System calls on Windows are also called native API calls or system services.

11. Apache has a built-in feature to do this logging.

Msg 1: $\langle \mathsf{req_i} \rangle$	Msg 3': $\langle \mathsf{resp_i} \rangle$	Msg 6: $\langle \mathsf{S_{pid_i}} \rangle$
Msg 2: $\langle \mathsf{req_i}, \mathsf{tag_i} \rangle$	Msg 4: $\langle \mathsf{tag_i}, \mathsf{pid_i} \rangle$	Msg 7: $\langle \mathsf{tag_i}, \mathsf{Sys_i} \rangle$
Msg 3: $\langle \mathsf{tag_i}, \mathsf{pid_i} \rangle$	Msg 5: $\langle \mathsf{pid_i} \rangle$	Msg 8: $\langle \mathsf{resp_i} \rangle$

$\mathsf{req_i}$	The ith client request
$\mathsf{tag_i}$	The unique tag for $\mathsf{req_i}$
$\mathsf{pid_i}$	The ID of the process/thread that serves $\mathsf{req_i}$
$\mathsf{resp_i}$	The response to $\mathsf{req_i}$
$\mathsf{S_k}$	The syscall sequence for process ID k in kernel
$\mathsf{Sys_i}$	The syscall sequence for $\mathsf{req_i}$ (a subsequence of $\mathsf{S_{pid_i}}$)

Fig. 3. Content of each internal message when processing a client request $\mathrm{req_i}$.

thread ID, the logger forwards them to the controller (explained in Section 5.1.3).

Note that this tag and the ID cannot be trusted, in that a compromised Apache process could send a forged tag or ID to the logger. Clearly, sending a tag that the proxy never inserted or the ID of a non-Apache process/thread can be detected readily. Similarly, changing the tag or process/thread ID to that of another existing request will result in a detection by associating multiple tags or IDs with one request (and none with another request). So, the best an attacker can do is to swap the tags or the IDs of two requests. However, this does not give an attacker any advantage in maintaining a large behavioral distance to the system-call sequences emitted by the other replica, since the attacker can issue any system calls from a compromised replica in the attacks we consider here.

5.1.3 Controller

For each `http` request, the controller first receives a log from the logger containing a tag and a process/thread ID, and then sends a request to the system-call hook to ask for system-call information for that ID. Upon receiving the system-call information, it locates the system-call subsequence that corresponds to the processing of the request (explained in the next paragraph) and sends it to the proxy along with the tag. Fig. 3 shows the content of each message exchanged among various components for a client request $\mathrm{req_i}$. Communications among the logger, the controller, and the proxy are via UNIX pipes or sockets.

Locating the system-call subsequence for each request is not easy because each process/thread processes one request after another, and system-call information from the kernel may cover multiple requests. One way is to use temporal information, e.g., instructing Apache and the kernel to log the time when a request is received and the time when each system call is made. However, Apache logs time at too coarse a granularity to enable us to accurately match system calls with requests, and this problem remained even after we modified the Apache source to log the most precise timing information supported by the operating system.

So, we take a different approach. We analyze the Apache source code to identify the last instruction in processing a request. We then insert a short piece of assembly code (one line), which does nothing but makes a special system call.[12] This special system call tells the controller when the processing of a request finishes, and helps the controller break a long system-call sequence into subsequences precisely at the end of the processing of each `http` request.

5.1.4 Implementation Issues

Thread ID on Windows. Instead of logging the *thread ID*, Apache on Windows logs the integer value of the *thread handle*. Although both the thread ID and the thread handle have a one-to-one correspondence with a thread on Windows, thread handles are only useful within the execution context of the program, and cannot be used to identify the thread from outside of the program. So, we modify the Apache source to log the thread ID instead, which requires only one line of code changed. We believe that this is, in fact, a bug in the Apache source. A bug report has been filed and Apache developers agreed that this is a bug.

Piped log. Apache provides a piped log to transfer the logs to another program (the logger in our system) instead of a plaintext log file. This works well on Linux using a UNIX pipe. However, there are problems on Windows, as more than one instance of the logger is invoked by Apache. Although the logger still works as expected, we believe there is a small performance penalty due to this problem on the Windows replica.

5.2 Game Server Implementation

The game server we use for our evaluation is the Peekaboom game server (see Section 4.3.2). Usually there are more than 1,000 player logins to the Peekaboom game server per day; on busy days, there could be as many as 20,000 logins. Each player spends roughly 25 minutes per login on average.

5.2.1 Game Events

The Peekaboom server utilizes a request handling model different from that of the Apache web server. Instead of assigning an isolated process or thread to process each request as in the Apache web server, it uses a single thread to process nearly all *game events* for different players. A game event is an object representing an action from a player (e.g., mouse clicking to reveal parts of an image or typing a guess) or the consequence of such an action (e.g., typing the correct guess results in a game event that ends the current game).

A player request may generate multiple responses. For example, a guess from Peek generates three events: a *guess event* to be processed by the game server to see if the guess is correct, two *new game events* to both players if the guess is correct, or two *guess resolve events* to the players if the guess is incorrect. Some game events are not triggered by any messages from the players, e.g., a timeout event is generated by the timer on the game server. Due to these complexities, the request-response transaction model used in the Apache system does not work well here. Instead, we measure behavioral distance between the system-call sequences for processing game events.

This difference between the definitions of system-call sequences for which behavioral distance is calculated in the web server and game server case studies highlights a potential challenge in applying our technique to other services. Specifically, in our experience, defining these

12. On Linux, we use the same system-call number that was used for sending commands from the controller to the system-call hook (see Section 5.1.1), with a different parameter. On Windows, we use a new system call that has not been implemented.

system-call sequences requires some understanding of the semantics of the service and the protocols used.

5.2.2 Logger and Controller

Since the Peekaboom server itself does not provide the necessary logging feature as in the Apache web server, we implement it as a shared library loaded by the game server using Java Native Interface (JNI). As in the Apache system, we need to attach a tag to every game event, so that the proxy is able to find system-call sequences for the same game event on different replicas. This turns out to be different from the case of Apache because the Peekaboom game server uses a single thread to process game events for all players. Therefore, process/thread IDs cannot help to differentiate system calls for processing different game events. However, we can use the player ID in conjunction with the game event type, which are available in the original Peekaboom server source code. Therefore, the proxy does not have to insert additional information to the messages to and from the players.

The logger also makes a special system call before and after the processing of every game event to indicate the start and end of the processing. This is the primary reason why the logger uses JNI: making system calls is platform dependent, and is best implemented in languages like C or C++ instead of Java.

The controller in the Peekaboom system is similar to that for Apache.

5.2.3 Critical Path of Server Responses

Our implementation for Peekaboom does not place behavioral distance measurement on the critical path of server responses. This is because of the complexity of the game server. In order to have behavioral distance measurement on the critical path, we need to precisely define the server responses' dependencies on game events. However, in the case of the Peekaboom game server, a response may be the result of zero or multiple messages from the players and many game events. It is too complex to define such dependencies precisely. Therefore, we choose not to associate the result of individual behavioral distance measurement with any particular server response, and to simply set off an alarm and tear down the game connections when any results of the behavioral distance measurements exceed the predefined threshold.

5.2.4 Implementation Issues

As required in most replicated systems, we take a number of steps to eliminate nondeterminism in the server replicas. Although determinism is a requirement for many replicated systems (c.f., [49]) and this difficulty is not unique in our architecture, it is a challenge that needs to be considered in any application of our architecture.

First, there are random number generators, e.g., to randomly select an image for the game. In order to make both replicas generate the same "random" numbers, we change the source of the game server to use the same fixed seed.

Second, when both players in a game are sending messages to the server, the server behavior may depend on the sequence in which the two messages are received. This turns out to be a problem because even if the proxy forwards the message from one player to both replicas first and then forwards the message from the other player, the two replicas may still receive the two messages in different orders (e.g., because the different network delays on the socket connections[13]). We found that this problem occurs in at least two scenarios: one is when the two players request to start a game at about the same time, and the other is when the two players are in a bonus game (to see what a bonus game is, please refer to www.peekaboom.org for details). To solve this problem, we associate a server acknowledgment with every message from a player. (Most of the player messages are already associated with server acknowledgments in the original program. We just need to add acknowledgments for messages sent in the above scenarios.) With the acknowledgments, the proxy ensures that a message from a player is forwarded to the replicas only after all acknowledgments for previous messages from the player's partner have been received. This results in some additional delay in server responses.

Third, the behavior of certain Java classes is not deterministic. For example, the sequence in which objects are returned by the `getNext()` method is not defined for the `Iterator` of a `HashSet` object. The Peekaboom game server uses a `HashSet` object for matching players in a game. It first puts all new players in a pool, which is a `HashSet` object, and then matches players in the pool by calling the `getNext()` method of the `Iterator` object of the pool. Since objects may be returned in different orders, players could be matched differently on the replicas. We solve this problem by replacing the `HashSet` object with a `LinkedHashSet` object, which returns objects in the sequence in which they were added. (Note that the sequence in which players are added to the pool is deterministic once the change explained in the previous paragraph is applied.)

Fourth, the game server updates the amount of idle time a player should wait before giving up. Such update messages are sent before and after a game starts, and the amount of idle time depends on the local clock of the game server, which is not the same for different replicas. There are a few ways to fix this, including synchronizing the clocks on replicas. We choose to apply a simple fix, instead, to remove the update message and let the client use its default setting (8 seconds) for the timeout. This simple fix turns out to work well without sacrificing any important features of the game server.

The above four issues require modifying or adding 13 lines of code in the original Peekaboom server source. Including the changes we made to attach a tag to every game event as explained in Section 5.2.2 (32 lines), we have modified less than 1 percent of the Peekaboom source.

6 Performance Overhead of Our New Architecture

In this section, we evaluate the performance overhead of the replicated web server and the replicated online game server we have implemented using recorded traces.

13. For each active player, there is one socket connection between the player and the proxy, and one socket connection between the proxy and each replica.

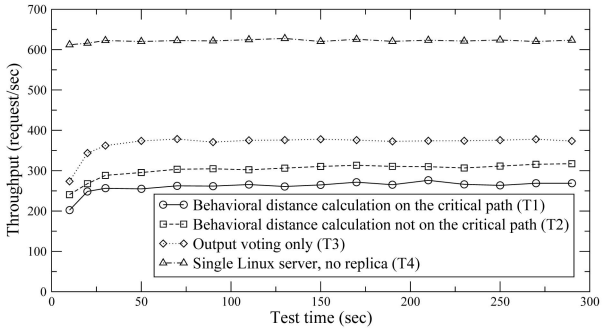


Fig. 4. Throughput of the web server.

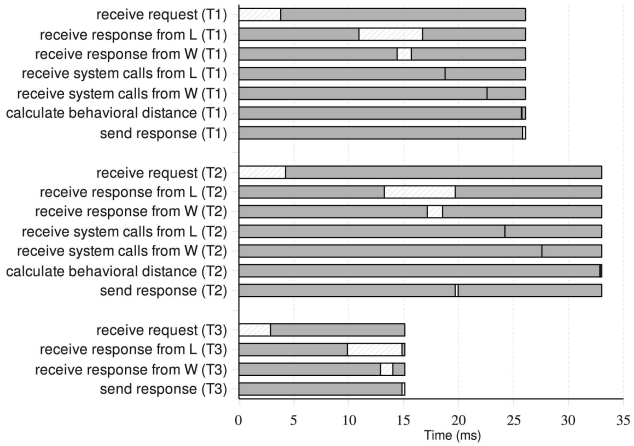


Fig. 5. Average latency measured by proxy.

Since we use virtual machines, only one computer is required. We use a Dell PowerEdge 2800 with two 3.2-GHz Intel Xeon CPUs with Hyper-Threading enabled. It has 8 Gbytes of memory and two SCSI hard drives in a RAID 1 configuration. The host computer is running Linux with a 2.6.15 Symmetric Multiprocessing (SMP) kernel. The host is connected to clients via an isolated LAN. VMware Workstation 5.5.2 is used to run two virtual machines as the replicas.

Both VMs are configured with two virtual CPUs, 2 Gbytes of virtual memory and a 15-Gbyte virtual SCSI hard drive. One runs Linux with a 2.6.15 SMP kernel, and the other runs Windows Server 2003 Enterprise Edition with Service Pack 1. A virtual gigabit switch connects the two virtual machines and the host.

6.1 Web Server

A typical way of evaluating the performance of a web server is to measure the throughput when the server is fully loaded. The recorded traces we use for this evaluation is the same as used in Section 4.3.1, i.e., the traces consist of more than two million requests for static pages on www.cylab.cmu.edu.

We first perform a number of tests with varying numbers of concurrent clients, to measure the throughput of the Apache web server in the nonreplicated setting. Results show that once the number of concurrent clients exceeds 10, further increasing this number will not improve the overall throughput. When there are virtual machines running, less than 10 concurrent clients are sufficient to fully load the system, but we choose to simulate 10 of them for all other tests.

Next, we perform four tests to evaluate our system in different configurations. In the first test (T1), the system measures both output voting and behavioral distance on the critical path of server responses. This is the configuration with the best security property, and at the same time gives the largest overhead on both throughput and latency because responses are forwarded to the clients after output voting and behavioral distance measurement finish. In the second test (T2), we do not perform behavioral distance measurement on the critical path. This should result in slightly better throughput and latency because responses are forwarded to the clients right after output voting is performed. Behavioral distance is not measured in the third test (T3). In the third test, we have a simple replicated system in which output voting is performed before responses are sent to the clients. The last test (T4) we do is to run the Apache web server directly on the host operating system without any replicated services.

Fig. 4 shows system throughput (requests/second) in the four tests. They show roughly a factor-of-2 cost in throughput (and latency, which is inversely proportional to the throughput) when providing the best security (T1), when compared with the results in a nonreplicated system (T4). Slightly better results are obtained when behavioral distance is not on the critical path of server responses (T2), or when the system utilizes output voting only (T3).

To better understand these results, we instrument the proxy to find what the system does during the lifetime of a request. The average results are shown in Fig. 5, where L and W denote the Linux and Windows replicas, respectively. We first compare the results of T1 and T2. Although in T2 responses are sent to the client earlier, messages from the replicas (including the `http` responses and the system-call information) have a longer delay in T2 than in T1. Ironically, this is because behavioral distance measurement is not on the critical path of server responses in T2, and the system continues to process new requests while measuring behavioral distances for previous requests. Given the same number of concurrent clients, at any time the servers in T2 will have less computation power allocated for each request. So, messages from the replicas appear to have longer delays.

Another interesting finding is that the replica running Linux spends longer time sending a response than the replica running Windows. Upon further investigation, it appears that the Linux web server tends to use smaller packet size and have more context switches among processes that are competing for the system resources. On Windows, server threads tend to finish sending all of their packets before giving up the system resources to other threads. Fig. 5 also confirms an earlier prediction (see Section 4) that caching behavioral distance results on the proxy is very effective, as we can see that behavioral distance calculation takes very little time on average in both T1 and T2.

6.2 Game Server

We perform a trace-driven evaluation, which we achieved by playing real recorded games on the Peekaboom server (see Section 4.3.2). In evaluating the performance overhead of the Peekaboom game server, we focus on the latency that players experience, as measured by the automatic player

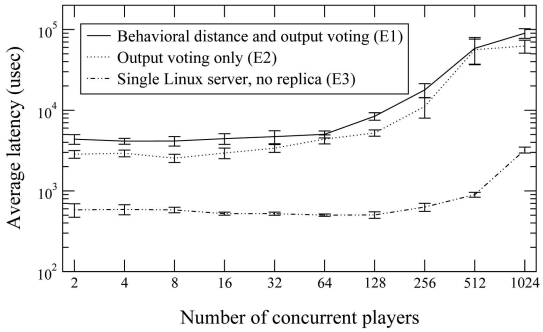


Fig. 6. Average latency measured by clients on the same LAN.

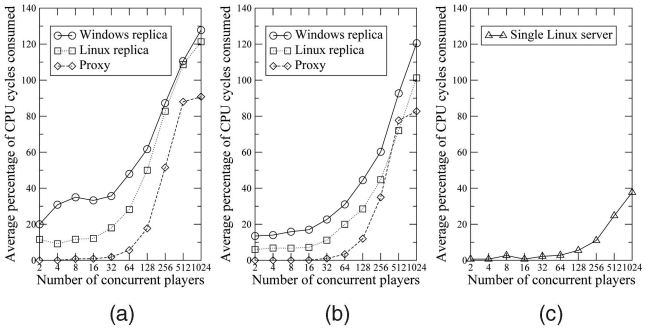


Fig. 7. Average CPU load of the replicas and the proxy. (a) E1. (b) E2. (c) E3.

program. Similar to what we did for Apache, we perform evaluations in three different system configurations. In the first configuration (E1), both behavioral distance measurement and output voting are performed to protect the online game servers. Note, however, that behavioral distance measurement is not on the critical path of server responses (see Section 5.2.3). In E2, only output voting is used. In E3, we only run the original Peekaboom game server on the host operating system without any virtual machines.

The latency measured by the automatic player program is the difference between when a message is sent and its acknowledgment is received. Each test includes at least 10 games, each of length 210 seconds, and uses a different number of players. The latency averages and standard deviations are shown in Fig. 6.

Results show that our system adds 3.5 to 8 ms to the latency when there are at most 128 concurrent players, which is hardly noticeable by humans. (The actual Peekaboom server usually has less than 80 concurrent players.) When the server is very busy, e.g., when there are 1,024 concurrent players, the players experience an extra 86-ms latency, which is still hardly noticeable. Also note that the results presented in Fig. 6 are latencies measured by a player program running on the same local area network of the server. A player over the Internet would also experience the round trip time to the server; if this is 100 ms or more, the extra latency our system adds to the user experience is about 8 percent when there are 128 users playing at the same time.

Fig. 7 shows the CPU load of the replicas and the proxy for the three tests, as reported by `top` on the host operating system. The CPU is not fully loaded in E1 and E2 until there are roughly 1,000 concurrent players, a number far larger than is typical for Peekaboom. Nevertheless, increased latencies in E1 and E2 result from the longer paths that requests travel in our replicated system.

7 Conclusion and Limitations

In this paper, we have introduced a novel approach for measuring behavioral distance using a new type of HMM. We show that this new approach based on HMM detects intrusions with substantially greater accuracy than previous approaches. We also demonstrate the design and implementation of a novel architecture employing behavioral distance using a web server and a game server. With the first trace-driven evaluation of behavioral distance, we show that the proposed architecture using behavioral distance is practical for real servers and able to detect compromised servers with high accuracy and moderate overhead.

References

[1] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff, "A Sense of Self for Unix Processes," *Proc. IEEE Symp. Security and Privacy (S&P)*, 1996.

[2] A. Wespi, M. Dacier, and H. Debar, "Intrusion Detection Using Variable-Length Audit Trail Patterns," *Proc. Third Int'l Symp. Recent Advances in Intrusion Detection (RAID)*, 2000.

[3] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis," *Proc. IEEE Symp. Security and Privacy (S&P)*, 2001.

[4] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," *Proc. IEEE Symp. Security and Privacy (S&P)*, 2001.

[5] J. Giffin, S. Jha, and B. Miller, "Detecting Manipulated Remote Call Streams," *Proc. 11th USENIX Security Symp.*, 2002.

[6] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly Detection Using Call Stack Information," *Proc. IEEE Symp. Security and Privacy (S&P)*, 2003.

[7] D. Gao, M.K. Reiter, and D. Song, "On Gray-Box Program Tracking for Anomaly Detection," *Proc. 13th USENIX Security Symp.*, 2004.

[8] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B.P. Miller, "Formalizing Sensitivity in Static Analysis for Intrusion Detection," *Proc. IEEE Symp. Security and Privacy (S&P)*, 2004.

[9] J. Giffin, S. Jha, and B. Miller, "Efficient Context-Sensitive Intrusion Detection," *Proc. Symp. Network and Distributed System Security (NDSS)*, 2004.

[10] D. Gao, M.K. Reiter, and D. Song, "Gray-Box Extraction of Execution Graph for Anomaly Detection," *Proc. 11th ACM Conf. Computer and Comm. Security (CCS)*, 2004.

[11] K. Tan, J. McHugh, and K. Killourhy, "Hiding Intrusions: From the Abnormal to the Normal and Beyond," *Proc. Fifth Int'l Workshop Information Hiding*, Oct. 2002.

[12] D. Wagner and P. Soto, "Mimicry Attacks on Host-Based Intrusion Detection Systems," *Proc. Ninth ACM Conf. Computer and Comm. Security (CCS)*, 2002.

[13] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating Mimicry Attacks Using Static Binary Analysis," *Proc. 14th USENIX Security Symp.*, Aug. 2005.

[14] J. Giffin, S. Jha, and B. Miller, "Automated Discovery of Mimicry Attacks," *Proc. Ninth Int'l Symp. Recent Advances in Intrusion Detection (RAID)*, 2006.

[15] K. Shin and P. Ramanathan, "Diagnosis of Processors with Byzantine Faults in a Distributed Computing System," *Proc. 17th Int'l Symp. Fault-Tolerant Computing (FTC)*, 1987.

[16] R.W. Buskens and R.P. Bianchini Jr., "Distributed On-Line Diagnosis in the Presence of Arbitrary Faults," *Proc. 23rd Int'l Symp. Fault-Tolerant Computing (FTC '93)*, June 1993.

[17] L. Alvisi, D. Malkhi, E. Pierce, and M.K. Reiter, "Fault Detection for Byzantine Quorum Systems," *IEEE Trans. Parallel Distributed Systems*, vol. 12, no. 9, Sept. 2001.

[18] L. Lamport, "The Implementation of Reliable Distributed Multiprocess Systems," *Computer Networks*, vol. 2, 1978.

[19] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, vol. 22, no. 4, Dec. 1990.

[20] M.K. Reiter, "Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart," *Proc. Second ACM Conf. Computer and Comm. Security (CCS '94)*, Nov. 1994.

[21] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM Trans. Computer Systems,* vol. 20, no. 4, Nov. 2002.
[22] C. Cachin and J.A. Poritz, "Secure Intrusion-Tolerant Replication on the Internet," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* 2002.
[23] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating Agreement from Execution for Byzantine Fault Tolerant Services," *Proc. 19th ACM Symp. Operating System Principles (SOSP '03),* Oct. 2003.
[24] M. Abd-El-Malek, G.R. Ganger, G.R. Goodson, M.K. Reiter, and J.J. Wylie, "Fault-Scalable Byzantine Fault-Tolerant Services," *Proc. 20th ACM Symp. Operating System Principles (SOSP '05),* Oct. 2005.
[25] D. Gao, M.K. Reiter, and D. Song, "Behavioral Distance for Intrusion Detection," *Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID),* 2005.
[26] D. Gao, M.K. Reiter, and D. Song, "Behavioral Distance Measurement Using Hidden Markov Models," *Proc. Ninth Int'l Symp. Recent Advances in Intrusion Detection (RAID),* 2006.
[27] P.H. Sellers, "On the Theory and Computation of Evolutionary Distances," *SIAM J. Applied Math.,* vol. 26, 1974.
[28] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," *Proc. IEEE Symp. Security and Privacy (S&P),* 1999.
[29] S. Cho and S. Han, "Two Sophisticated Techniques to Improve HMM-Based Intrusion Detection Systems," *Proc. Sixth Int'l Symp. Recent Advances in Intrusion Detection (RAID),* 2003.
[30] 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," *Proc. 15th USENIX Security Symp.,* Aug. 2006.
[31] L. Cavallaro, "Comprehensive Memory Error Protection via Diversity and Taint-Tracking," PhD dissertation, Universita' Degli Studi Di Milano, 2007.
[32] L.R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proc. IEEE,* Feb. 1989.
[33] X.D. Hoang, J. Hu, and P. Bertok, "A Multi-Layer Model for Anomaly Intrusion Detection Using Program Sequences of System Calls," *Proc. 11th IEEE Int'l Conf. Networks (ICON),* 2003.
[34] I.M. Meyer and R. Durbin, *Comparative ab initio Prediction of Gene Structures Using Pair HMMs.* Oxford Univ. Press, 2002.
[35] L. Pachter, M. Alexandersson, and S. Cawley, "Applications of Generalized Pair Hidden Markov Models to Alignment and Gene Finding Problems," *Computational Biology,* vol. 9, no. 2, 2002.
[36] J. Just, J. Reynolds, L. Clough, M. Danforth, K. Levitt, R. Maglich, and J. Rowe, "Learning Unknown Attacks—A Start," *Proc. Fifth Int'l Symp. Recent Advances in Intrusion Detection (RAID),* 2002.
[37] J. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich, "The Design and Implementation of an Intrusion Tolerant System," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* 2002.
[38] E. Totel, F. Majorczyk, and L. Me, "COTS Diversity Based Intrusion Detection and Application to Web Servers," *Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID),* 2005.
[39] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the Detection of Anomalous System Call Arguments," *Proc. Eighth European Symp. Research in Computer Security (ESORICS),* 2003.
[40] Sufatrio and R.H.C. Yap, "Improving Host-Based IDS with Argument Abstraction to Prevent Mimicry Attacks," *Proc. Eighth Int'l Symp. Recent Advances in Intrusion Detection (RAID),* 2005.
[41] G. Tandon and P. Chan, "Learning Rules from System Calls Arguments and Sequences for Anomaly Detection," *Proc. ICDM Workshop Data Mining for Computer Security (DMSEC),* 2003.
[42] G. Tandon and P. Chan, "Learning Useful System Call Attributes for Anomaly Detection," *Proc. 18th Int'l Florida Artificial Intelligence Research Symp. (FLAIRS),* 2005.
[43] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow Anomaly Detection," *Proc. IEEE Symp. Security and Privacy (S&P),* 2006.
[44] C. Parampalli, R. Sekar, and R. Johnson, "A Practical Mimicry Attack against Powerful System-Call Monitors," *Proc. ACM Symp. Information, Computer and Comm. Security (ASIACCS '08),* Mar. 2008.
[45] S. Chen, J. Xu, E.C. Sezer, P. Gauriar, and R.K. Iyer, "Non-Control-Data Attacks are Realistic Threats," *Proc. 14th USENIX Security Symp.,* Aug. 2005.
[46] L.E. Baum and T. Petrie, "Statistical Inference for Probabilistic Functions of Finite State Markov Chains," *Ann. Math. Statistics,* vol. 37, 1966.
[47] R.I.A. Davis, B.C. Lovell, and T. Caelli, "Improved Estimation of Hidden Markov Model Parameters from Multiple Observation Sequences," *Proc. 16th Int'l Conf. Pattern Recognition (ICPR),* 2002.
[48] L. von Ahn, R. Liu, and M. Blum, "Peekaboom: A Game for Locating Objects in Images," *Proc. Conf. Human Factors in Computing Systems (CHI),* 2006.
[49] P. Narasimhan, L.E. Moser, and P.M. Melliar-Smith, "Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications," *Proc. 18th IEEE Symp. Reliable Distributed Systems (SRDS '99),* pp. 263-273, Oct. 1999.

Debin Gao received the MS and PhD degrees in computer engineering from Carnegie Mellon University in 2004 and 2006, respectively. He is currently an assistant professor in the School of Information Systems, Singapore Management University. He was previously a systems software engineer in CyLab, Carnegie Mellon University. His research interests include computer and network security. He received the Ann and Martin McGuinn Graduate Fellowship for the year 2005, and the Frank J. Marshall Graduate Fellowship for the year 2004.

Michael K. Reiter received the BS degree in mathematical sciences from the University of North Carolina, Chapel Hill (UNC-CH), in 1989 and the MS and PhD degrees in computer science from Cornell University in 1991 and 1993, respectively. He is currently the Lawrence M. Slifkin distinguished professor in the Department of Computer Science, UNC-CH. He has held technical leadership positions in both the industry and academe. From 1998 to 2001, he was the director of Secure Systems Research, Bell Laboratories. From 2001 to 2007, he was a professor and the technical director of CyLab, Carnegie Mellon University. His research interests include computer and communications security and distributed computing. He is a senior member of the IEEE Computer Society.

Dawn Song received the PhD degree in computer science from the University of California, Berkeley (UC Berkeley) in 2002. She is an assistant professor in the Computer Science Division, UC Berkeley. Her research interests include security and privacy issues in computer systems and networks. She is the author of more than 60 research papers in areas ranging from software security, networking security, database security, distributed systems security, to applied cryptography. She is the recipient of various awards and grants including the US NSF CAREER Award, the IBM Faculty Award, the George Tallman Ladd Research Award, the Sloan Award, and the Best Paper Award in USENIX Security Symposium.