

**Beyond Output Voting:
Detecting Compromised Replicas using Behavioral Distance**

Debin Gao, Michael K. Reiter, Dawn Song

December 6, 2006
CMU-CyLab-06-019

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Beyond Output Voting: Detecting Compromised Replicas using Behavioral Distance

Debin Gao
debin@cmu.edu

Michael K. Reiter
reiter@cmu.edu

Dawn Song
dawnsong@cmu.edu

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. In this paper we present a novel architecture to detect mimicry attacks using “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. We detail the design and implementation of our architecture, which takes advantage of virtualization to achieve its goals efficiently. We apply our system to implement intrusion-tolerant web and game servers, and through trace-driven simulations demonstrate that our approach can achieve low false-alarm rates and moderate performance costs even when tuned to detect stealthy mimicry attacks.

Keywords: behavioral distance, intrusion detection

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 [10, 31, 29, 24, 15, 9, 12, 8, 16, 11]. 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* [27, 30, 19, 17] that evade detection by virtually all such models. In mimicry attacks, 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 [26, 4, 3] or masks [20, 23, 21, 6, 5, 32, 1] Byzantine faults or in-

trusions 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 attacks, e.g., to leak sensitive data or to attack other machines in the network.

Behavioral distance [13, 14] has been proposed to detect carefully crafted mimicry attacks that would evade detection by a host-based anomaly detector or output voting. This approach compares the low-level (e.g., system call) behaviors of two diverse replicas when processing the same, potentially malicious, inputs. Assuming that the two replicas are diverse and vulnerable only to different exploits, a successful attack on one of them should induce a detectable increase in the behavioral distance. This makes mimicry attacks substantially more difficult, because to avoid detection, the behavior of the compromised process must be close to the behavior of the uncompromised one. Behavioral distance goes beyond output voting to measure the similarity of server behaviors, instead of the similarity in server outputs.

In this paper, we present the design, implementation and evaluation of a novel architecture to detect mimicry attacks using behavioral distance. Whereas our prior work focused on algorithms for computing behavioral distance, here we address the systems issues necessary to make this technique practical. We present a complete architecture based on virtualization for monitoring the system call behaviors of diverse replicas on the same computer, 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 two types of servers: 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, much of the game server’s processing is decoupled from individual requests, and its responses are not in one-to-one correspondence with client requests; this makes it necessary to pair the low-level behav-

iors 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.

The evaluation we perform is, to our knowledge, the first trace-driven evaluation of behavioral distance; whereas the cursory evaluations in our previous work utilized synthetic web workloads, here we utilize recorded workloads of production web and game server deployments to evaluate the detection accuracy and performance of our web and game servers. We show, for example, that our web server using behavioral distance, when configured to detect the “best” mimicry attacks, yielded as few as 3 false alarms when processing a recorded workload of over 2 million client requests, which is a false alarm rate about an order of magnitude lower than previous results [14] on average. Similarly configured, our game server yielded 14 false alarms when processing 39,000 recorded game events. We also describe an alternative behavioral distance calculation particular to the game server that reduces the false alarm rate to near zero while retaining the ability to detect the type of mimicry attacks against which we perform our evaluation. In terms of performance, the web server’s throughput drops to about 50% compared to a standalone web server on the same physical machine, and players experience an overhead of 8 to 86 milliseconds (msecs) in additional latency for the game server with 128 to 1024 concurrent players.

2 Related Work

Behavioral distance was introduced to evaluate the extent to which processes—potentially running different programs and executing on different platforms—behave similarly in response to a common input [13]. The first calculation of behavioral distance was inspired by evolutionary distance [25], though it was shown to be inferior to a subsequent proposal using Hidden Markov Models (HMMs) [14]. These prior works focused on the algorithms for calculating behavioral distance, and evaluated these algorithms using synthetic simulations of static web page retrievals. Neither detailed a practical system for realizing behavioral distance measurement generally, or reported trace-driven analyses of its accuracy and performance (or on servers other than a web server), as we do here.

N-variant systems [7] 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. The system we propose here, instead, uses behavioral distance to detect potentially unforeseen types of compromises of one of two off-the-shelf servers.

Numerous systems have employed output voting to detect some types of server compromises. For example, the HACQIT system [18, 22] uses two web servers, Microsoft’s 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 [28]. 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 all behaviors (system calls) of the replicas, and is applicable to virtually any services (not just web servers).

3 System Architecture

There are at least three components in a system that utilizes behavioral distance—two replicas and a proxy. The replicas run servers, either on different operating systems or with programs of different code bases. The proxy serves as a gateway between the replicas and the clients.

Our architecture hosts the replicas and proxy on a single physical machine, using virtualization. One benefit of doing so is that network delays for messages between the replicas and the proxy can be minimized. When implemented as virtual machines, these delays are limited only by the speed of memory copies. Since there are at least three messages exchanged between each replica and the proxy for every client request (the request forwarded from the proxy to the replica, the response from the replica, and the system call information from the replica), this savings can be significant. Another advantage is that resources can be better managed among the proxy and the replicas. This resource sharing is handled by the scheduler on the host operating system automatically; if the proxy and replicas were running on different computers, available CPU cycles or memory on one could not be used by others. Using virtual machines also reduces the hardware and maintenance costs of the system.

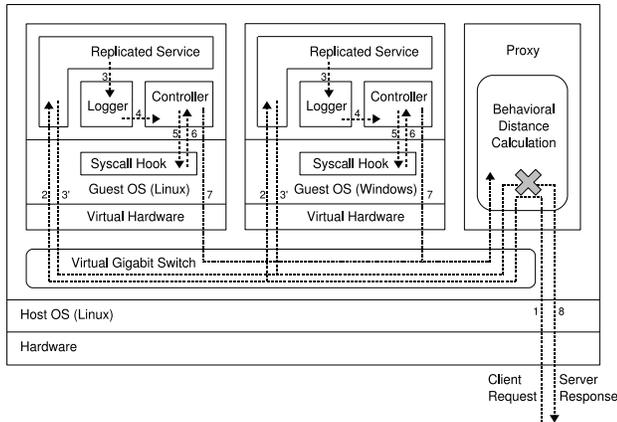
For these reasons, virtualization is attractive for implementing replicated systems that measure behavioral distance.

Below we outline the system structure, and then explain the details with two concrete examples—behavioral distance for a web server and an online game server.

3.1 General System Structure

There are generally two approaches to setting up the replicas and the proxy. One is to configure the host machine with three guest operating systems, each running on an isolated virtual machine. This setup allows each virtual machine to have a fair share of the system resources on the host machine. A second approach is to configure the host with only two virtual machines. In such a setup, the proxy runs on the host operating system directly.

We choose the second approach for two reasons. First, the proxy plays a different role in the system from the replicas. The proxy connects to both the clients and the replicas, while the replicas are required to talk only to the proxy. The other reason is that the second setup performs faster than the first setup according to our experiments. This is partly because the first setup imposes another operating system between the hardware and the proxy, which consumes noticeable resources. In using the second approach, we choose Linux as the host operating system.



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

Figure 1. Architecture of the system

Figure 1 shows the system architecture and the messages involved in a client request and response. The lifetime of a client request and the corresponding response is as follows. Upon receiving a request from the client (Message

1), the proxy forwards it (Message 2) to both replicas after some necessary modifications (these modifications are discussed in Section 3.2.2). A replicated server processes the request and sends its response (Message 3') back to the proxy. At the same time, the replicated server also sends a log (Message 3) containing important information about the processed request to the logger, which forwards the log (Message 4) to the controller. The controller processes the log, requests (Message 5) and receives (Message 6) system call information for the corresponding request, and forwards the system call information (Message 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 proxy forwards the response (Message 8) to the client. The proxy also maintains a cache that remembers the results of behavioral distance calculations for system call sequences it has seen.

3.2 Web Server Implementation

In this section, we detail how we have applied this architecture to protect Apache web servers serving http requests. The two replicas in this system run Apache httpd on a Linux and a Windows operating system, respectively. The Apache web server is a multi-process application on Linux and a multi-threaded application on Windows. A process/thread is assigned to each http request and is responsible for processing that request. Our system measures the behavioral distance between the system calls of the corresponding process and thread that serve the same request.

3.2.1 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 call¹ is used for a user program running as root (the controller, see Section 3.2.3) to send commands to the kernel to start/stop system call interception and to request system call numbers 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 call² hooking is implemented as a kernel driver, which locates and overwrites the KiSystemService table. The KiSystemService table contains the addresses of all system call handling functions. By overwriting

¹We utilize system call numbers that are reserved but not implemented yet on the 2.6.15 Linux kernel.

²System calls on Windows are also called native API calls or system services.

ing them with addresses of new system call handling functions, system call information can be extracted. The new system call handling functions simply save the system call numbers in the kernel, and then invoke the original system call handling functions. Unlike the case of Linux, Windows provides an interface for a user program to send requests to and receive responses from a kernel driver. Therefore we do not have to implement a new system call to do this.

3.2.2 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 when the server is heavily loaded, there could be many requests from clients, which are being processed simultaneously by different processes/threads; therefore, simply using the timing information would not reliably match system call sequences with their corresponding requests/responses. To do this matching in a reliable way, 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/responses.

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, the insertion of this tag does not result in much additional overhead. After inserting the tag, we modify the configuration file to instruct Apache to log the value of the tag as well as the process ID of the process (or the thread ID of the thread) that served the request, and send this information to the logger. Upon receiving the tag and the process/thread ID, the logger simply forwards it to the controller, which is explained in the next section.

Note that we have to implement the logger as a separate program instead of a component of the controller because the logger is instantiated by Apache, whereas the controller has to start its execution before Apache starts up.

3.2.3 Controller

The controller is the most intelligent component in a replica. For each `http` request, it first receives a log from the logger (which contains the tag and the process/thread ID), and then sends a request to the system call hook in the kernel to ask for the system call information for that process/thread ID. Upon receiving the system call information, it locates the subsequence that corresponds to the processing of the request and sends it to the proxy along with the tag. Figure 2 shows the content of each message exchanged among various components for a client request req_i . Communications among the logger, the controller and the proxy are via UNIX pipes or sockets.

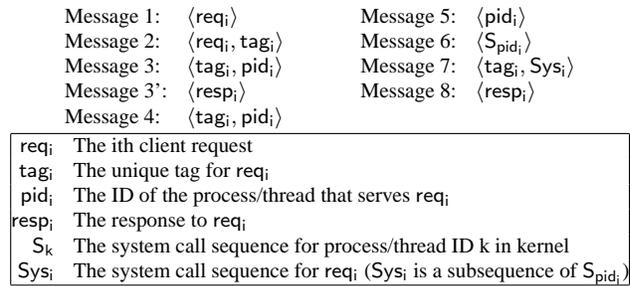


Figure 2. Content of each internal message when processing a client request req_i

When the web server is heavily loaded, a process/thread will be processing one request after another; therefore, the controller needs to break the long system call sequence for each process/thread into shorter pieces, such that each piece corresponds to the processing of an `http` request.

One way to do this is to rely on temporal information. E.g., we can instruct Apache to log the time when a request is received, and instruct the system call hook to record the time when each system call is made. However, we find that this is not a reliable way because the timing information provided by Apache and the operating system is not precise enough. E.g., Apache only logs up to seconds, which is far from the precision we require. We also tried modifying the Apache source to log the most precise timing information provided by the operating system. However, many system calls are still made “at the same time” because they are made between two consecutive hardware time interrupts.

We decide to take a more reliable and more precise 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 call³. This special system call tells the controller when the processing of a request finishes, and helps the controller to break a long system call sequence into subsequences precisely at the end of the processing of each `http` request.

3.3 Game Server Implementation

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 the sense that each transaction consists of a single request and a response. In this section, we show another system in

³On Linux, we use the same system call number that was used for sending commands from the controller to the system call hook (see Section 3.2.1), with a different parameter. On Windows, we use a new system call that has not been implemented.

which behavioral distance is used to protect an online 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. The fact that server responses are dynamically generated also make it more complex, when compared to simple web servers in which most responses are static `html` pages.

The online game server we choose to work with is the Peekaboom game server (www.peekaboom.org). Peekaboom [2] 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. 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.

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 4.3) provide a realistic view of the suitability of our approach to similar services written in C/C++, for example.

3.3.1 Game events

The Peekaboom server utilizes a request handling model different from the Apache web server. Instead of assigning an isolated process/thread to process each request as in the Apache web server, the Peekaboom game server uses a single thread to process nearly all *game events* from 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 of a guess) or the consequence of such an action (e.g., the consequence of typing a correct guess is a game event that ends the current game).

A player request may generate zero or 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* sent back to both players if the guess is correct, or two *guess resolve events* sent back 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 for behavioral distance measurement does not work well here. Instead, we measure behavioral distance between the system call sequences for processing game events.

3.3.2 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 JNI (Java Native Interface). 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 separate system calls for processing different game events. However, we can use the player ID in conjunction with the game event type as the tag. Since the player ID and event type are available in the original Peekaboom server source code, we do 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 of that game event. This is the primary reason why the logger is integrated with the main server using JNI: making system calls is not platform independent, and is best implemented in languages like C or C++ instead of Java.

The controller in the Peekaboom system works very similarly as in the Apache system.

3.3.3 Implementation issues

Nondeterminism As behavioral distance is best measured when the replicas are performing the same tasks, and to accommodate output voting in addition to behavioral distance measurement, we take a number of steps to eliminate nondeterminism in the server replicas.

First, there are random number generators, e.g., to randomly select an image for the game, and to randomly select a label for an image (there are multiple valid labels for every image). 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 and then the message from the other player, the two replicas may still receive the two messages in different orders (e.g., because the different message sizes and different network delays on the socket connections⁴). We found that this problem occurs in at least two scenarios: one is when the two players

⁴For each active player, there is one socket connection between the player and the proxy, and one socket connection between the proxy and each replica.

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 acknowledgement with every message from a player. (Most of the player messages are already associated with server acknowledgements in the original program. We just need to add acknowledgements for messages sent in the above scenarios.) With the acknowledgements, the proxy ensures that a message from a player is forwarded to the replicas only after all acknowledgements for 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 on the replicas, players are matched differently. 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 simply 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 3.3.2 (32 lines), we have modified less than 1% of the Peekaboom source.

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 multi-

ple 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 simply set off an alarm and tear down the game connections when any results of the behavioral distance measurements exceed the predefined threshold.

4 Evaluation and Discussion

In this section, we evaluate the two systems we have implemented, i.e., the replicated web server and the replicated online game server. We want to see how well our systems behave in detecting carefully crafted mimicry attacks [27, 30, 19, 17]. We will consider the same type of mimicry attack as in our previous work [12, 13, 14], in which the attacker tries to make a system call `open` followed by a system call `write`, as this is seemingly the least the attacker must do to modify or create data on the server machine. We also evaluate the performance overhead of the systems when detecting these attacks.

4.1 Hardware and software configuration

Since we use virtual machines, only one physical computer is required. The computer we use is a Dell PowerEdge 2800 with two Intel Xeon CPUs running at 3.2 GHz each with Hyper-Threading enabled. It has 8 GB of memory and two SCSI hard drives in a RAID 1 configuration. The host computer is running the Linux operating system with a 2.6.15 SMP (Symmetric Multiprocessing) kernel. In both systems (the web server and the online game server), the host is connected to clients via an isolated local area network. VMware Workstation 5.5.2 is used to create and run two virtual machines as the replicas.

Both virtual machines are configured with two virtual CPUs, 2 GB of virtual memory and a 15 GB virtual SCSI hard drive. One of them runs the Linux operating system with a 2.6.15 SMP kernel, and the other runs Windows Server 2003 Enterprise Edition with Service Pack 1. A virtual gigabit switch is created to connect the two virtual machines and the host.

4.2 Web Server

We want to see how the system behaves when serving real web traffic instead of traffic simulated by a benchmarking tool as in previous projects [13, 14]. The trace we use consists of a five-month-long log of client requests for static pages on the public web server of CyLab (www.cylab.cmu.edu). This five-month-long data consists of more than 2 million requests on about 2,700 distinct URLs, including `html` pages, images, videos and etc.

The behavioral distance measurement for this system follows the HMM approach [14]. In this approach, a training set is used to build the HMM, a validation set is used to detect overfitting the training data, and a testing set is used to evaluate the accuracy of the model. The training set contains a subset (of a size that varies per experiment; see Section 4.2.1) of the 2,700 distinct URLs. We request each URL in this subset once, and use the system call sequences induced to build the HMM. The validation set consists of 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 dataset excluding the validation set. Both replicas run Apache `httpd 2.2.2`.⁵

4.2.1 Detection accuracy

To evaluate the detection accuracy, we measure the number of false alarms generated when the threshold of behavioral distance is set to detect the “best” mimicry attack. A mimicry attack [27, 30, 19, 17] is one of the most powerful attacks against an intrusion detection system, in which it is assumed that the attacker has a copy of the model used by the anomaly detector. The attacker analyzes the model and executes its attacks in a way that induces behaviors (a system call sequence) that the model does not distinguish from normal. In the case of HMM-based behavioral distance, the distance threshold can always be set to detect a mimicry attack; the only question is what false alarm rate does that setting induce? To evaluate this, we compute the estimated best mimicry attack for our HMM (see [14]) in the cases where the exploitable vulnerability is on Linux or Windows. In each case, we set the threshold of the system to detect this mimicry, and then measure the number of false alarms the system generates when processing the testing set.

We perform this test a few times, by setting the size of the training data to be certain percentages of the distinct requests. This is to simulate the scenario in which when new contents are added to a web server, the system administrator may not want to re-train the behavioral distance model. Therefore, the training set may not contain all the distinct requests. Figure 3 shows the number of false alarms when the training set consists of 40% to 100% of the distinct requests, when the system is tested on about 2 million requests recorded in 150 days.

From the results we can see that our system is able to detect software intrusions with very high accuracy. In particular, our system generates only 3 false alarms in processing more than 2 million requests, when the training set consists of all distinct requests. When some requests are not included in the training set, the number of false alarms increases to about 60, which is still very good. These results

⁵Apache on Linux and Apache on Windows are different code bases.

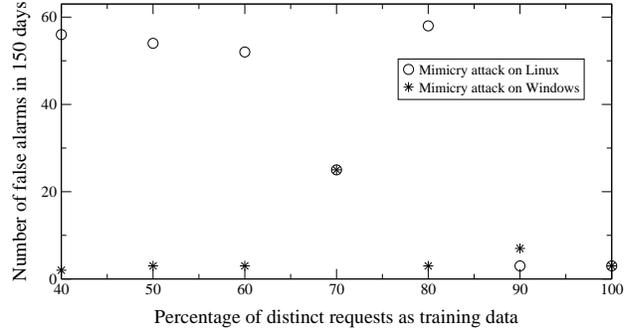


Figure 3. Number of false alarms when detecting the “best” mimicry attack

are also about an order of magnitude better than those previously reported [13, 14]. From these results, it is recommended that the model is re-trained when the training set consists of less than 90% of the distinct requests, if very low false-alarm rate is desired.

4.2.2 Performance overhead

A typical way of evaluating the performance overhead of a web server is to measure the throughput when the server is fully loaded. In order to fully load the web server, we simulate concurrent clients. Figure 4 shows the throughput of the Apache web server with varying number of concurrent clients, when the Apache web server is the only service running on our host computer, i.e., when there is no virtual machine running. We can see that once the number of concurrent clients exceeds ten, further increasing the number of concurrent clients will not improve the overall throughput. When there are virtual machines running, less than ten concurrent clients are sufficient to fully load the system, but we choose to simulate ten of them for all other tests.

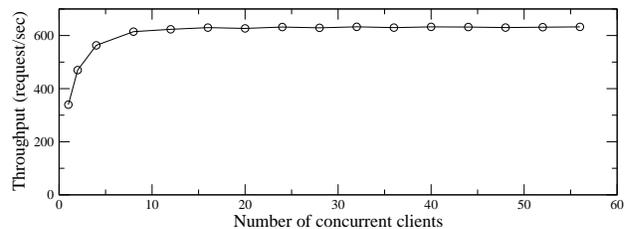


Figure 4. Throughput of the web server with different numbers of concurrent clients

We perform four tests to evaluate our system in different configurations. The first test (T1) we perform is to measure 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 client 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 client 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 client. The last test (T4) we do is to run the Apache web server directly on the host operating system without any replicated services.

Figure 5 shows the overall throughput of the system in all the four tests, when throughput is measured in terms of the total number of requests processed per second. Table 1 shows the average latency measured by the clients on the same local area network.

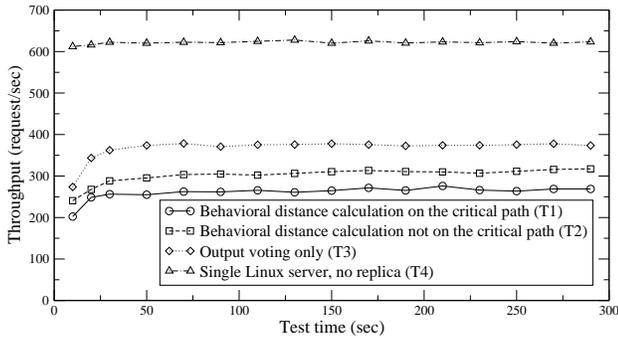


Figure 5. Throughput of the web server

	T1	T2	T3	T4
Average latency (msec)	38.48	33.30	27.33	16.09

Table 1. Average latency measured by clients

Figure 5 and Table 1 show that we lose the throughput and latency by a factor of about 2 when providing the best security property (T1), when compared with the results in a non-replicated system (T4). Slightly better throughput and latency 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).

In order to better understand the system in the first three tests, we instrument the proxy to find what the system does from the time a request enters the system until it leaves. The average results are shown in Figure 6, where L and W denote the replicas running Linux and Windows, 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

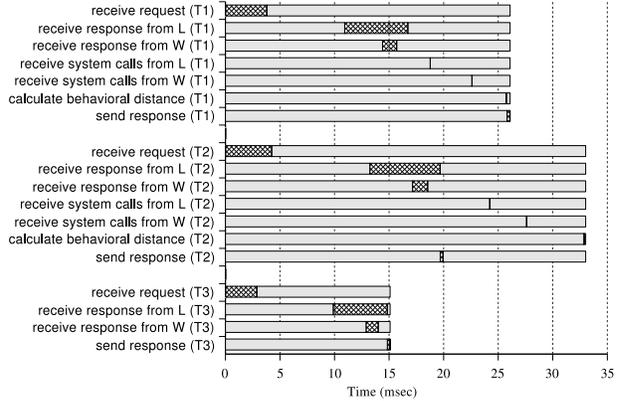


Figure 6. Average latency measured by proxy

information) appear to 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. The system continues to process new requests while measuring behavioral distances for previous requests. So at any time in T2, the server has a higher workload, in the sense that it not only processes current requests, but performs behavioral distance measurement for previous requests. So, messages from the replicas appear to have longer delays.

Another interesting finding is that the replica running Linux spends longer 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.

Figure 6 also confirms earlier predictions [14] 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.

4.3 Game Server

Again we want to perform a trace-driven evaluation, which we achieved by playing real recorded games on the Peekaboom game server. 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 (pseudorandomly; see Section 3.3.3). 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.

4.3.1 Detection accuracy

To evaluate the detection accuracy of behavioral distance on the Peekaboom game server, we take a similar approach as in the Apache system. We run the system as described in Section 3.3 and (randomly select and) replay the recorded games. System calls made for processing each game event are collected on both replicas. We collected system call sequences for a total of 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.

In our tests, an HMM is built using the training and validation sets [14]; the threshold of the system is set to detect the estimated best mimicry attack; and the model is then evaluated on the testing set. During the evaluation, we recorded 14 false alarms (the same number of false alarms are recorded for mimicry attacks on Linux and Windows) for over 39,000 game events in the testing set. Note that these results were obtained when we use the same HMM for 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 processing function. This request parsing event is special in that we expect it 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 will almost certainly be treated as malformed by the uncompromised replica. In this case, the attack system calls must have a small behavioral distance with those produced by only the request parsing event on the uncompromised replica: if the attack generates other events on the compromised replica, 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, provided that the proxy checks that the two replicas perform the same types of game events, and maintains the set of system calls that is 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 made possible because we are able to obtain fine-grained event type information from the Peekaboom game server on both replicas. This is non-trivial especially when replicas

are running different code bases. Another limitation of our analysis is that we have considered only one type of mimicry attack, albeit one (`open` followed by `write`) that is seemingly the minimum an attacker must do to modify or create data on the system being protected.

4.3.2 Performance overhead

In evaluating the performance overhead of the Peekaboom game server, we focus on the latency that players experience. Since a single connection between a player and the proxy is used throughout the game for each player, the proxy program does not have enough information to measure this latency; therefore we measure the latency from the automatic player program.

Similar to what we did for the Apache system, 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 3.3.3. 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 defined as the difference between the time when a message is sent and the time the corresponding acknowledgement is received. We run a few tests, each with a different number of concurrent players. In each test, at least ten games, each of length 210 sec, are played and the average results and their standard deviations are presented in Figure 7.

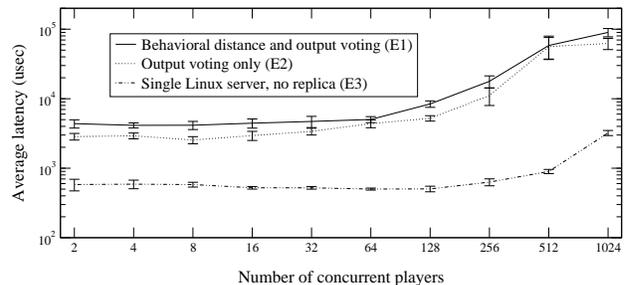


Figure 7. Average latency measured by clients on the same LAN

Results show that our replicated system adds 3.5 to 8 milliseconds to the latency when there are less than (or equal to) 128 concurrent players, which is hardly noticeable by human beings. (The actual Peekaboom server usually has less than 80 concurrent players.) When the server is very busy, e.g., when there are 1024 concurrent players, the players experience an additional 86-millisecond latency, which is still hardly noticeable. Also note that the results pre-

sented in Figure 7 are latencies measured by an automatic player program running on the same local area network of the server. A human player over the Internet would also experience the round trip time to the server machine, which is typically over 100 msec,⁶ which means the additional latency our system adds to the end-user experience is about 8% when there are 128 users playing at the same time.

Figure 8 shows the CPU load of the replicas and the proxy for the three tests reported by `top` on the host operating system. Results show that the CPU resource is not the bottleneck in most cases. (Only when there are 1024 concurrent players does the system become almost fully loaded in E1 and E2.) The latency in E1 and E2 when there are less than 128 concurrent players is mainly due to network delays — packets need to travel a much longer path. The increase in latency when there are more than 128 concurrent players is because of the threading model used by the Peekaboom server, in which a single thread is used to process game events for all players. We suggest using a multi-threaded model if this latency needs to be reduced.

5 Conclusion

In this paper, we presented a novel architecture to detect compromised replicas using behavioral distance. Our system monitors the system-call behaviors of diverse replicas to detect mimicry attacks that can evade detection by output voting on a replicated system or host-based anomaly detection system on an isolated computer. We detail the design and implementation of intrusion-tolerant web and game servers using our architecture, and show that we can achieve low false-alarm rates and moderate performance overhead when detecting very stealthy mimicry attacks.

References

- [1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, October 2005.
- [2] L. Ahn, R. Liu, and M. Blum. Peekaboom: a game for locating objects in images. In *Proceedings of the 2006 Conference on Human Factors in Computing Systems (CHI 2006)*, 2006.
- [3] L. Alvisi, D. Malkhi, E. Pierce, and M. K. Reiter. Fault detection for Byzantine quorum systems. *IEEE Transactions on Parallel Distributed Systems*, 12(9), September 2001.
- [4] R. W. Buskens and R. P. Bianchini, Jr. Distributed on-line diagnosis in the presence of arbitrary faults. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, June 1993.
- [5] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 2002.
- [6] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), November 2002.
- [7] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems – A secretless framework for security through diversity. In *Proceedings of the 15th USENIX Security Symposium*, August 2006.
- [8] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, 2004.
- [9] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.
- [10] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, 1996.
- [11] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graph for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer & Communication Security (CCS 2004)*, 2004.
- [12] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th USENIX Security Symposium*, 2004.
- [13] D. Gao, M. K. Reiter, and D. Song. Behavioral distance for intrusion detection. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, 2005.
- [14] D. Gao, M. K. Reiter, and D. Song. Behavioral distance measurement using Hidden Markov Models. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID 2006)*, 2006.
- [15] J. Giffin, S. Jha, and B. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

⁶We measured the RTT between a server on our department network (`www.ece.cmu.edu`) and `www.google.com`. Results were between 108 msec and 119 msec in 20 runs.

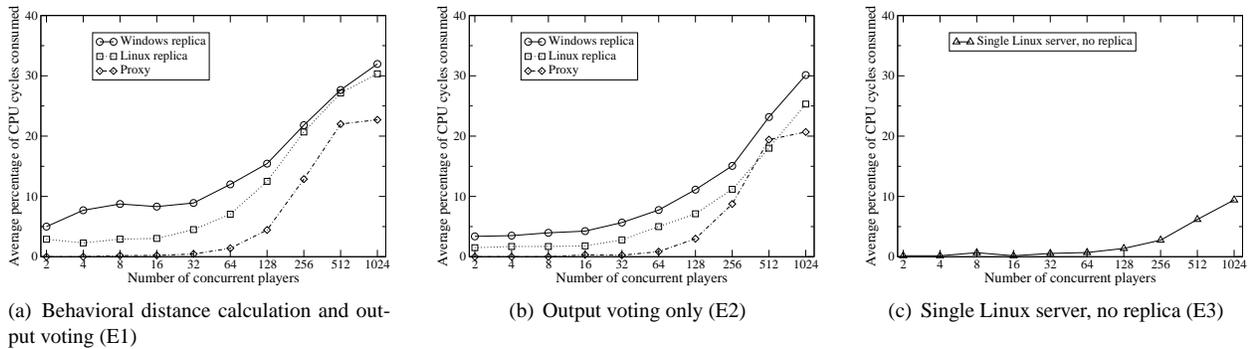


Figure 8. Average CPU load of the replicas and the proxy

- [16] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of Symposium on Network and Distributed System Security*, 2004.
- [17] J. Giffin, S. Jha, and B. Miller. Automated discovery of mimicry attacks. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID 2006)*, 2006.
- [18] J. Just, J. Reynolds, L. Clough, M. Danforth, K. Levitt, R. Maglich, and J. Rowe. Learning unknown attacks - A start. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, 2002.
- [19] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.
- [20] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2, 1978.
- [21] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, November 1994.
- [22] J. Reynolds, J. Just, E. Lawson, L. Clough, and R. Maglich. The design and implementation of an intrusion tolerant system. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN02)*, 2002.
- [23] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), December 1990.
- [24] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.
- [25] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26, 1974.
- [26] K. Shin and P. Ramanathan. Diagnosis of processors with Byzantine faults in a distributed computing system. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, 1987.
- [27] K. Tan, J. McHugh, and K. Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Proceedings of the 5th International Workshop on Information Hiding*, October 2002.
- [28] E. Totel, F. Majorczyk, and L. Me. Cots diversity based intrusion detection and application to web servers. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, 2005.
- [29] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.
- [30] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, 2002.
- [31] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Proceedings of the 2000 Recent Advances in Intrusion Detection*, 2000.
- [32] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.