Gray-Box Anomaly Detection using System Call Monitoring

by

Debin Gao

A dissertation submitted

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

Carnegie Mellon University

Pittsburgh Pennsylvania, USA

Jan 2007

Thesis committee

Prof. Raj Rajkumar (Carnegie Mellon University)

Prof. Michael K. Reiter (Carnegie Mellon University), Chair

Prof. R. Sekar (Stony Brook University)

Prof. Dawn Song (Carnegie Mellon University)

© 2007 Debin Gao

Abstract

Many host-based anomaly detection systems monitor a process by observing the system calls it makes, and comparing these calls to a model of normal behavior for the program that the process is executing. In this thesis we explore two novel approaches for constructing the normal behavior model for anomaly detection.

We introduce *execution graph*, which is the first model that both requires no static analysis of the program source or binary, and conforms to the control flow graph of the program. When used as the model in an anomaly detection system monitoring system calls, it (i) accepts only system call sequences that are consistent with the control flow graph of the program; (ii) is maximal given a set of training data, meaning that any extensions to the execution graph could permit some intrusions to go undetected. We formalize and prove these claims, and evaluate the performance of an anomaly detector using execution graphs.

Behavioral distance compares the behavior of a process to the behavior of another process that is executing on the same input but that either runs on a different operating system or runs a different program that has similar functionality. Assuming their diversity renders these processes vulnerable only to different attacks, a successful attack on one of them should induce a detectable increase in the "distance" between the behavior of the two processes. We propose two black-box approaches for measuring behavioral distance, the first inspired by evolutionary distance and the second using a new type of Hidden Markov Model.

We additionally build and evaluate a replicated system, which uses behavioral distance

to protect Internet servers. Through trace-driven evaluations we show that we can achieve low false-alarm rates and moderate performance costs even when the system is tuned to detect very stealthy mimicry attacks.

Acknowledgements

During the course of this thesis I have accumulated a great debt of gratitude to many people:

My advisors, Mike Reiter and Dawn Song, who have given their guidance, help and encouragement through the years. This research would not have been possible without the support of them. Their impact on my research has been tremendous and invariably positive. I am incredibly fortunate to have had the opportunity of working closely with them.

My faithful committee members Raj Rajkumar and R. Sekar. Thank you for your willingness to serve on my committee and your comments which have helped me improve the work of this thesis.

The staffs in ECE and CyLab, who have always been available in these year whenever I need assistance. Thanks to Linda Whipkey, Helen Borek-Conti, and many others.

And last but certainly not the least, my family for their immense love and thoughtful encouragement. Mum, Dad and Dele, thank you for believing in me and always doing your best to lend a helping hand whenever I need it.

Table of Contents

1 Introduction **1**

1.1 Execution Graphs 3

1.2 Behavioral Distance 4

2 Related Work **8**

3 Execution Graphs **12**

3.1 Execution Graphs 12

3.2 Control Flow Graphs 19

3.3 Properties of Execution Graphs 27

3.3.1 Well-behaved executions 28

3.3.2 Properties of Execution Graphs 29

3.4 Performance evaluation 42

4 Behavioral Distance **44**

4.1 System call phrases 46

4.2 ED-based Behavioral Distance 47

4.2.1 Learning the Distance table 50

4.2.1.1 Initializing the Distance table 51

4.2.1.2 Iteratively updating the Distance table 54

4.2.2 Real-time monitoring 55
4.2.3 Parameter settings 55
4.3 HMM-based Behavioral Distance 56
4.3.1 Elements of the HMM 57
4.3.2 Computing $Pr_\lambda([S_1, S_2])$ 60
4.3.3 Building λ 63
4.3.3.1 Refining a_i 64
4.3.3.2 Refining b_i 67
4.3.4 Implementation issues 69
4.4 Detection accuracy of ED-based and HMM-based Behavioral Distance . . . 70
4.4.1 Estimating the best mimicry 71
4.4.2 False-alarm rate when detection the "best" mimicry 76
4.5 Design and implementation of intrusion-tolerant web and game servers . . . 77
4.5.1 System Architecture 78
4.5.1.1 General System Structure 79
4.5.1.2 Web Server Implementation 81
4.5.1.3 Game Server Implementation 84
4.5.2 Evaluation and Discussion 89
4.5.2.1 Hardware and software configuration 89
4.5.2.2 Web Server 89
4.5.2.3 Game Server 95

5 Conclusion **101**
5.1 Limitation of system-call-based anomaly detection techniques 102
5.2 Future work 103

List of Figures

3.1 Source code and execution graph of Example 3.1.1 17

3.2 Control flow graph of Example 3.1.1 22

3.3 Source code and control flow graph of Example 3.2.1 26

3.4 Execution graph of Example 3.2.1 27

4.1 Example of two nucleotide sequences 48

4.2 Example of system call sequences observed on two replicas 48

4.3 Architecture of the system 80

4.4 Content of each internal message when processing a client request $\mathsf{req_i}$ . . . 83

4.5 Number of false alarms when detecting the “best” mimicry attack 91

4.6 Throughput of the web server with different numbers of concurrent clients . 92

4.7 Throughput of the web server 93

4.8 Average latency measured by proxy 94

4.9 Average latency measured by clients on the same LAN 98

4.10 Average CPU load of the replicas and the proxy 100

List of Tables

3.1 Performance overhead for processing system calls 42

4.1 False-alarm rate when detecting the estimated-best mimicry attack 78

4.2 Average latency measured by clients 93

Chapter 1

Introduction

Numerous attacks on software systems result in a process' execution deviating from its normal behavior. Prominent examples include code injection attacks on server processes, resulting from buffer overflow and format string vulnerabilities. A significant amount of research has sought to detect such attacks through monitoring the behavior of the process and comparing that behavior to a model of "normal" behavior. These techniques are also called *anomaly detection* techniques because, in contrast to signature-based detection, deviations from the normal behavior are taken as indications of intrusions. The behavior that is considered in most recent research projects is the sequence of system calls made by the process. This is mainly because system calls are the gateway between user processes and the operating system kernel, and a process presumably is able to affect its surroundings primarily through system calls.

Many host-based intrusion detection systems (e.g., [FHSL96, Pro03, Wag99, WD01]) and related sandboxing and confinement systems (e.g., [PFH03, WLAG93]) monitor the system calls emitted by a process in order to detect deviations from a previously constructed model of system call behavior. We coarsely divide these systems into "black box", "gray box" and "white box" approaches, based on the information they use to build the model

to which they compare system calls at run time. On the one hand, black-box and gray-box methods build a model of system-call behavior by monitoring sample executions. Within this space, black-box detectors employ only the system call number (and potentially the arguments, though we do not consider arguments in this thesis) that pass through the system call interface when system calls are made (e.g., [FHSL96, TMK02]). A gray-box detector extracts additional runtime information from the process, e.g., by looking into the process' memory (e.g., [FKF+03, SBDB01]). On the other hand, white-box approaches obtain the model by statically analyzing the source code or binary (e.g., [FGH+04, GJM02, GJM04, WD01]).

By their nature, black-box and gray-box detectors detect *anomalous* behavior, i.e., behavior different from "normal" runs, regardless of whether it results from an intrusion or an execution path that was not encountered during training. In contrast, white box detectors detect actual deviations from the program text, for which an intrusion is virtually the only conceivable explanation (assuming that the program is not self-modifying). As such, white-box detectors can be designed to have a zero false-positive rate, in the sense that an alarm always indicates an intrusion. Since minimizing the number of false positives is a significant factor in gaining user acceptance, this is an important advantage of white-box approaches.

White-box approaches, however, are not always viable. First, source code is often not available, and the complexity of performing static analysis on, e.g., x86 binaries is well documented.[1] Static analysis is also difficult for programs protected by obfuscation or digital rights management (DRM) technologies that are designed to render static analysis

[1] This complexity stems from difficulties in code discovery and module discovery [RVL+97], with numerous contributing factors, including: variable instruction size (Prasad and Chiueh claim that this renders the problem of distinguishing code from data undecidable [PC03a]); hand-coded assembly routines, e.g., due to statically linked libraries, that may not follow familiar source-level conventions (e.g., that a function has a single entry point) or use recognizable compiler idioms [SDA02]; and indirect branch instructions such as `call/jmp reg32` that make it difficult or impossible to identify the target location [PC03b, RVL+97]. Due to these issues and others, binary analysis/rewrite tools for the x86 platform have strict restrictions on their applicable targets [Lu99, PC03a, RVL+97, SDA02].

of control flow all but impossible (e.g., [CTL98]). Finally, white-box techniques cannot detect rare execution paths which, in some cases, are indications of error states resulting from an attack. We thus believe that examination of gray-box and black-box approaches can play an important role where white-box approaches are unavailable.

In this thesis we explore two novel approaches for constructing the normal behavior model for anomaly detection using gray-box and black-box techniques. The approaches considered here introduce new ways of constructing the normal behavior model, and offer properties for host-based anomaly detection that were not offered in prior techniques. In particular, the execution graph is the best one can achieve in approximating white-box models of control-flow behavior without the risk of accepting intrusions, and the behavioral distance approach forms a basis for more sensitive host-based intrusion detection in service architectures employing diverse server replicas.

1.1 Execution Graphs

We present a new gray-box model, called an *execution graph*, that is, to our knowledge, the first gray- or black-box technique for which a positive relationship to what is achievable via common white-box techniques can be proved analytically. Intuitively, the goal that we set for our technique is to build a model that accepts the same system call sequences as would be accepted by a model built from the control flow graph of the program, which is the basis of many white-box techniques. This, of course, is not achievable, since our gray-box technique can train only on observed runs of the program, which may miss entire branches of the program that static analysis would uncover. Nevertheless, using gray-box techniques alone, our approach constructs an execution graph with the following two useful properties: First, the system call sequences (the language) accepted by the execution graph are a subset of those accepted by the control flow graph of the program. Second, the language accepted by the execution graph is *maximal* for the training sequences it was provided. Specifically,

we show that there exists a program of which the control flow graph would accept the same language as the execution graph. In other words, if the execution graph were to accept any other system call sequence s, then there is a program that can emit exactly the same training sequences but for which the control flow graph would not accept s.

In some sense, this is the best one can hope to achieve toward using a gray-box technique to mimic the power of a control flow model obtained via white-box analysis. Moreover, as the control flow model that we set for our goal is equivalent to the most restrictive white-box models known in the literature—the model is sensitive not only to the sequence of system calls, but the sequence of active function calls when each system call occurs—our approach mimics some of the best white-box techniques known today, using only gray-box analysis. Additionally, we demonstrate through a prototype implementation that monitoring via an execution graph is very efficient.

1.2 Behavioral Distance

We present a new approach for detecting the anomalous behavior of a process, in which the model of "normal" is a "replica" of the process running in parallel with it, operating on the same inputs. At a high level, our goal is to detect any behavioral deviation between replicas operating on the same inputs, which will then indicate that one of the replicas has been compromised. As we will show, this approach will better detect *mimicry attacks* [WS02, TMK02, KKM+05, GJM06] than previous approaches. A mimicry attack is one in which the injected attack code masquerades as the original software program so that the anomaly detector cannot differentiate execution of the attack code from execution of the original software program. In addition, our approach has an immediate application in fault-tolerant systems, which often run replicas and compare their responses (not behavior) to client requests to detect (e.g., [SR87, BB93, AMPR01]) or mask (e.g., [Lam78, Sch90, CRL03, YMV+03]) faults. When considering attacks, it is insufficient

to simply compare the responses to detect faults, because certain intrusions may not result in observable deviation in the responses (but may nevertheless go on to attack the interior network, for example). Our method of detecting behavioral deviation between replicas can significantly improve the resilience of such fault-tolerant systems by detecting more stealthy attacks.

Monitoring for deviations between replicas would be a relatively simple task if the replicas were identical. However, in light of the primary source of attacks with which we are concerned—i.e., software faults and, in particular, code injection attacks that would corrupt identical replicas identically—it is necessary that the "replicas" be as diverse as possible. We thus take as our goal the task of measuring the behavioral distance between two diverse processes, be they distinct implementations of the program (e.g., as in n-version programming [CA78]), the same implementation running on different platforms (e.g., one Linux, one Windows), or even distinct implementations on diverse platforms. In this thesis, we propose a method to measure behavioral distance between replicas and show that our method can work with competing, off-the-shelf, diverse implementations without modification.

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

A key observation in our work, however, is that even though the system call sequences might not be similar in any syntactic way, they will typically be correlated in the sense

that a particular system call subsequence emitted by one replica will usually coincide with a subsequence emitted by the other replica (but one that is syntactically very different). These correlations could be determined either through static analysis of the replica executable (and the libraries), or by first subjecting the replicas to a battery of well-formed (benign) inputs and observing the system call sequences induced coincidentally. The former is potentially more thorough, but the latter is more widely applicable, being unaffected by difficulties in static analysis of binaries for certain platforms or, in the future, of software obfuscated to render static analysis very difficult for the purposes of digital rights management (e.g., [CTL98]). So, we employ the latter method here.

We propose two black-box approaches for calculating the behavioral distance between two processes when the behavior of each process is the system calls it emits. The first approach is inspired by evolutionary distance (ED) [Sel74], and the second approach is by using a new type of Hidden Markov Model. Through an empirical evaluation of this measure using three web servers on two different platforms (Linux and Windows), we demonstrate that both approaches hold promise for better intrusion detection with moderate overhead. Since an HMM offers a more powerful mathematical structure which better accounts for the order of system calls, it should able to detect intrusions with greater accuracy. This is confirmed in our experiments.

Additionally, we show the implementation and evaluation of a system using behavioral distance that makes it very difficult for an intrusion to evade detection. 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 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.

The evaluation we perform is, to our knowledge, the first trace-driven evaluation of behavioral distance; 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. 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.

The remainder of this thesis is organized as follows. Chapter 2 describes related work in this area. The execution graph model and behavioral distance approaches are discussed in Chapter 3 and Chapter 4, respectively. We conclude in Chapter 5.

Chapter 2

Related Work

We coarsely divide host-based intrusion detection systems into “black box”, “gray box” and “white box” approaches, based on the information they use to build the model to which they compare system calls at run time [GRS04b]. On the one hand, black-box and gray-box methods build a model of system-call behavior by processing sample executions. Within this space, black-box detectors employ only the system call number (and potentially the arguments) that passes through the system call interface when system calls are made. A gray-box detector extracts additional runtime information from the process, e.g., by looking into the process' memory. On the other hand, white-box approaches obtain the model by statically analyzing the source code or binary.

Black-box approaches were pioneered by Forrest et al. [FHSL96], who introduced an approach to characterize normal program behavior in terms of sequences of system calls. System call sequences are broken into patterns of fixed length, which are stored in a table. Wespi et al. [WDD00] extended this approach to permit variable-length patterns of system calls. To our knowledge, Sekar et al. [SBDB01] proposed the first gray-box approach, coupling the system call number with the program counter of the process when the system call is made. Feng et al. [FKF$^+$03] proposed extending the gray-box information used to

include return addresses on the call stack of the process when a system call is made. While the benefits and costs of many of these approaches have been studied [GRS04b], the behavior of none of these approaches has been formally related to that of the white-box system call models. In fact it is generally easy to confirm that these prior black- and gray-box models neither contain nor are contained by the white-box models, in terms of the languages of system call sequences they accept.

Numerous white-box approaches to intrusion detection have focused on monitoring a process' system-call conformance with the control flow graph of the program it is ostensibly running. One of the earliest works, due to Wagner and Dean [WD01], generates a range of models based on the control flow graph of the program via static analysis of the source. Their most accurate model resulted in very substantial runtime monitoring overheads. This cost, as well as the need for analyzing source code, were addressed in works due to Giffin et al. [GJM02, GJM04] and Feng et al. [FGH+04]. These works included modifying the binary to permit the runtime monitor to perform more efficiently. Abadi et al. [ABEL05] introduced Control-Flow Integrity (CFI) enforcement for Windows on the x86 architecture, which dictates that software execution must follow a path of a Control Flow Graph derived by static binary analysis. PAID (Program-semantics Aware Intrusion Detection) is a kernel-compiler patch to detect computer system intrusions [LLC06]. PAID automatically extracts system-call patterns of programs through parsing the source files and then uses the info to compare with the run-time system-call execution patterns of programs to detect intrusions.

A technique proposed to make mimicry attacks more difficult utilizes system-call arguments (e.g., [KMVV03, BCS06]). Models for detecting anomalous system calls typically monitor the system-call numbers but not their arguments, and so a mimicry attack can issue system calls that are consistent with the model but for which the arguments of certain calls are modified to be "malicious". To the extent that system-call arguments can be accurately modeled, this can increase the difficulty of mimicry attacks. While we do not

utilize system-call arguments in our work, it is potentially a way to augment the strength of our techniques.

N-variant systems [CEF+06] are closely related to our work on behavioral distance. 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 [JRC+02, RJL+02] 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 [TMM05]. 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).

The key to one of the techniques for behavioral distance we present here is a novel HMM construction. HMMs have been studied for decades and used in a wide variety of applications, owing to two features: First, HMMs are very rich in mathematical structure

and hence can form the theoretical basis for a wide range of applications. Second, when applied properly, HMMs work very well in practice for many important applications. One of the most successful applications of HMMs is in speech recognition [Rab89]. HMMs have also been used in intrusion detection systems, e.g., to model the system-call behavior of a single process [WFP99], and to model privilege flows [CH03]. 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 in behavioral distance.

Variations of ordinary HMMs might seem to be more suited to our needs. For example, "pair HMMs" [MD02] and "generalized pair HMMs" [PAC02] 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 behavioral distance, and have devised a custom solution, instead.

Chapter 3

Execution Graphs

In this chapter, we first describe what an execution graph is and how it is constructed. After that, we briefly define control flow graphs. The properties of the execution graphs are then discussed. Finally, we present our evaluation results for the execution graphs.

3.1 Execution Graphs

In this section we describe our model, called an *execution graph*, for anomaly detection, which is built using a gray-box technique. Our technique assumes that the program being monitored is implemented in a programming language for which the runtime utilizes a call stack, where each stack frame corresponds to a function call in the program and includes a return address. Every implementation of the C and C++ programming languages known to us satisfies this criteria, and these languages are the primary motivations for our work.

The execution graph technique we describe in this section works, during both training and monitoring, by observing system calls along with additional runtime information that it extracts upon each system call, namely the return addresses on the call stack of the monitored process when the system call is made. We define a system call along with the return addresses on the call stack when a system call is made as an *observation*. Each such

observation can be represented by an arbitrary-length vector of integers, each in the range of $[0, 2^{32})$ assuming a 32-bit platform. The last element of the vector is the system call number, and the preceding elements are the return addresses on the call stack when the system call is made, with the first address being an address in `main()`, i.e., an address in the first function executed.

We formally define the concept of observation below, and we call a sequence of observations an *execution*.

Definition 3.1.1 (Observation) *An* observation *is a tuple of positive integers* $\langle r_1, \ldots, r_k \rangle$*, where* $k > 1$*.* □

Definition 3.1.2 (Execution) *An* execution *is an arbitrary-length sequence of observations.* □

In particular, for an observation $\langle r_1, r_2, \ldots, r_k \rangle$, r_1 is an address in `main()`, r_{k-1} is the "return address" which corresponds to the instruction that makes the system call,[1] and r_k is the system call number.

We next introduce the concept of an execution graph, which is built by observing executions as defined above. The goal we set for this new model is to build a model that accepts the same system call sequences that will be accepted by most models built from white-box techniques. Informally, we need to extract function call structures from observations so that a graph similar to the control flow graph can be built. To achieve this we analyze every two consecutive system calls and the return addresses on the call stack when each system call is made, i.e., to analyze two consecutive observations. Since each return address represents a stack frame, consecutive observations reveal some information about the function call structure of the program. In the following definition, we show how this information is used to build an execution graph.

[1]Though on most platforms, system calls are implemented differently from function calls, r_{k-1} can be retrieved from the stack in a similar fashion, and we still refer to it as a return address.

The execution graph is one of the most important concepts in this section, especially the inductive definition of the edges in the graph. These edges will be comprised by three sets, E_{rtn}, E_{crs} and E_{call}. Intuitively, we use E_{rtn} to represent the returning of a function to its calling location, use E_{crs} to represent the execution flow within a function, and use E_{call} to represent the calling from a function call site to its call target. These three sets of edges are defined in the base case by processing consecutive observations. The inductive part of the definition is used to post-process these sets of edges, and to discover "missing" edges, where the relationship between two nodes could be derived from the executions, but not by processing any individual pair of observations. (This induction is further explained in an example after we formally present the definition.)

Definition 3.1.3 (Execution graph, leaf node, $\stackrel{\mathsf{crs}}{\rightarrow}$) *An* execution graph *for a set of executions $\mathcal{X}$ is a graph $\text{EG}(\mathcal{X}) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$, where V is a set of nodes, and E_{call}, E_{crs}, $E_{\mathsf{rtn}} \subseteq V \times V$ are directed edge sets, defined as follows:*

- *For each execution $X \in \mathcal{X}$ and each observation $\langle r_1, r_2, \ldots, r_k \rangle \in X$, V contains nodes $r_1, r_2, \ldots, r_k$. r_k is called a* leaf node *of the* execution graph $\text{EG}(\mathcal{X})$*. In the case where $\langle r_1, r_2, \ldots, r_k \rangle$ is the first observation in an execution, r_k is also denoted as an* enter *node; in the case where $\langle r_1, r_2, \ldots, r_k \rangle$ is the last observation in an execution, r_k is also denoted as an* exit *node. (Note that an execution graph could have more than one* enter *node and more than one* exit *node.)*

- *The sets E_{call}, E_{crs}, E_{rtn} are defined inductively to contain only edges obtained by the following rules:*

  - **(Base case)** *For each execution X in $\mathcal{X}$ and each pair of consecutive observa-*

tions $\langle r_1, r_2, \ldots, r_k\rangle$, $\langle r'_1, r'_2, \ldots, r'_{k'}\rangle$ *in* X,

$$\begin{aligned} E_{\mathsf{rtn}} &\leftarrow E_{\mathsf{rtn}} \cup \{(r_{i+1}, r_i)\}_{\ell \le i < k} \\ E_{\mathsf{crs}} &\leftarrow E_{\mathsf{crs}} \cup \{(r_\ell, r'_\ell)\} \\ E_{\mathsf{call}} &\leftarrow E_{\mathsf{call}} \cup \{(r'_i, r'_{i+1})\}_{\ell \le i < k'} \end{aligned}$$

where

$$\ell = \begin{cases} k-1 & \textit{if } \langle r_1, r_2, \ldots, r_k\rangle = \langle r'_1, r'_2, \ldots, r'_{k'}\rangle \\ \left(\arg\max_j : \langle r_1, r_2, \ldots, r_j\rangle = \langle r'_1, r'_2, \ldots, r'_j\rangle\right) + 1 & \textit{otherwise} \end{cases}$$

If r_k *is an* enter *node,*

$$E_{\mathsf{call}} \leftarrow E_{\mathsf{call}} \cup \{(r_i, r_{i+1})\}_{1 \le i < k}$$

If $r'_{k'}$ *is an* exit *node,*

$$E_{\mathsf{rtn}} \leftarrow E_{\mathsf{rtn}} \cup \{(r'_{i+1}, r'_i)\}_{1 \le i < k'}$$

– **(Induction)** *Define the relation* $r \xrightarrow{\mathsf{crs}} r'$ *to be true if there exists a path from* r *to* r' *consisting of only edges in* E_{crs}.

  * *If* $(x_0, x_1) \in E_{\mathsf{call}}$, $x_1 \xrightarrow{\mathsf{crs}} x_2$, *and* $(x_2, x_3) \in E_{\mathsf{rtn}}$, *then* $E_{\mathsf{rtn}} \leftarrow E_{\mathsf{rtn}} \cup \{(x_2, x_0)\}$ *and* $E_{\mathsf{call}} \leftarrow E_{\mathsf{call}} \cup \{(x_3, x_1)\}$;
  * *If* $(x_0, x_1) \in E_{\mathsf{call}}$, $x_1 \xrightarrow{\mathsf{crs}} x_2$, *and* $(x_3, x_2) \in E_{\mathsf{call}}$, *then* $E_{\mathsf{call}} \leftarrow E_{\mathsf{call}} \cup \{(x_3, x_1)\}$ *and* $E_{\mathsf{call}} \leftarrow E_{\mathsf{call}} \cup \{(x_0, x_2)\}$;
  * *If* $(x_1, x_0) \in E_{\mathsf{rtn}}$, $x_1 \xrightarrow{\mathsf{crs}} x_2$, *and* $(x_2, x_3) \in E_{\mathsf{rtn}}$, *then* $E_{\mathsf{rtn}} \leftarrow E_{\mathsf{rtn}} \cup \{(x_1, x_3)\}$ *and* $E_{\mathsf{rtn}} \leftarrow E_{\mathsf{rtn}} \cup \{(x_2, x_0)\}$.

□

Note that the integers in an observation serve as labels for the nodes created. For simplicity, we do not differentiate a node and its label, i.e., in the above definition, r and x denote both the nodes and their labels.

Example 3.1.1 illustrates the reason why such an inductive definition is necessary. (Source code and the execution graph of Example 3.1.1 are shown in Figure 3.1.)

Example 3.1.1 *In this example,* `f()` *is called twice from* `main()`*, while each time it is called not all instructions in* `f()` *are executed. Some execution paths of the program might not be uncovered, e.g., due to the fixed values of* `a` *and* `b` *in the executions. In this example, edges* `(f.3, main.5)` *and* `(f.5, main.3)` *can only be obtained by the inductive definition in Definition 3.1.3.*

With the inductive definition in Definition 3.1.3, the execution graph as shown in Figure 3.1 can be obtained even if the value of `a` *and* `b` *are fixed in executions* $\mathcal{X}$.[2] □

Definition 3.1.4 ($\stackrel{\text{call}}{\rightarrow}$, $\stackrel{\text{rtn}}{\rightarrow}$) *Let* $\text{EG}(\mathcal{X}) = (V,\ E_{\mathsf{call}},\ E_{\mathsf{crs}},\ E_{\mathsf{rtn}})$ *be an execution graph.* $r \stackrel{\text{call}}{\rightarrow} r'$ *iff there exists a path from* r *to* r' *consisting of only edges in* E_{call}. $r \stackrel{\text{rtn}}{\rightarrow} r'$ *iff there exists a path from* r *to* r' *consisting of only edges in* E_{rtn}. □

Recall that we want to mimic the power of the most restrictive control flow graph model known in the literature, where not only the system call sequence, but also the sequence of active function calls when each system call occurs, are captured. To do this, we use the notion of *execution stack* to capture the active function calls allowed in an execution graph.

Definition 3.1.5 ($\stackrel{\text{xcall}}{\rightarrow}$, Execution stack) *Let* $\text{EG}(\mathcal{X}) = (V,\ E_{\mathsf{call}},\ E_{\mathsf{crs}},\ E_{\mathsf{rtn}})$ *be an execution graph.* $r \stackrel{\text{xcall}}{\rightarrow} r'$ *iff*

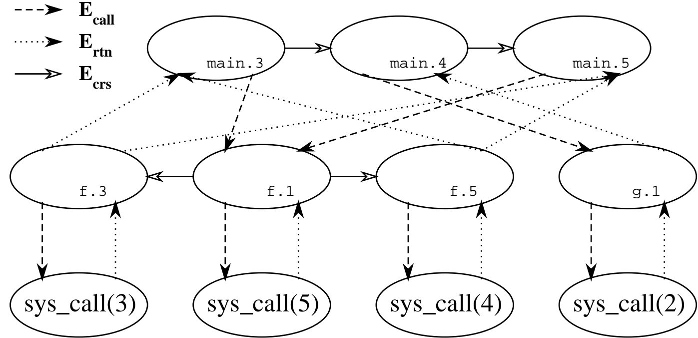
[2] Nodes in an execution graph are typically denoted by integers only. In Figure 3.1 we show the correspondence with the line number and function name, for the purpose of illustration.

```
    int main(int argc, char *argv[]) {
1:     int a, b;
2:     a = 1; b = 2;
3:     f(a);
4:     g();
5:     f(b);
    }

    void f(int x) {
1:     sys_call(5);
2:     if (x == 1)
3:        sys_call(3);
4:     else if (x == 2)
5:        sys_call(4);
    }

    void g() {
1:     sys_call(2);
    }
```

(a) source code of Example 3.1.1



(b) execution graph of Example 3.1.1

Figure 3.1: Source code and execution graph of Example 3.1.1

- $(r, r') \in E_{\mathsf{call}}$*; or*
- *There exists a node* $r'' \in V$*, such that* $(r, r'') \in E_{\mathsf{call}}$ *and* $r'' \stackrel{\mathsf{crs}}{\to} r'$.

An execution stack *in* EG$(\mathcal{X})$ *is a sequence of nodes* $\langle r_1, r_2, \ldots, r_n \rangle$*, such that*

- *For each* $1 \leq i < n, r_i \stackrel{\mathsf{xcall}}{\to} r_{i+1}$*; and*
- r_1 *corresponds to an address in* `main()`*, i.e., an address in the first function executed; and*
- r_n *is a leaf node.*

□

Intuitively, an execution stack captures a system call and the active functions (functions that have not returned) when the system call is made, which is also what an observation captures. However, an execution stack might or might not have a corresponding observation in the executions $\mathcal{X}$ that are used to construct the execution graph $\textsc{eg}(\mathcal{X})$.

We next define the notion of *successor*. Intuitively, if observation x' follows another observation x in an execution, then x' corresponds to an execution stack that is a *successor* of the execution stack corresponding to x. The notion of *successor* in an execution graph defines whether a system call (and the corresponding active functions) are allowed to follow another system call.

Definition 3.1.6 (Successor) *Execution stack $s' = \langle r'_1, \ldots, r'_{n'} \rangle$ is a* successor *of execution stack $s = \langle r_1, \ldots, r_n \rangle$ in an execution graph $\textsc{eg}(\mathcal{X}) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$ if there exists an integer k such that $r_n \stackrel{\mathsf{rtn}}{\rightarrow} r_k$, $(r_k, r'_k) \in E_{\mathsf{crs}}$, $r'_k \stackrel{\mathsf{call}}{\rightarrow} r'_{n'}$, and for each $1 \leq i < k$, $r_i = r'_i$.* □

Definition 3.1.7 (Execution path) *An* execution path *δ is a sequence of execution stacks in an execution graph $\textsc{eg}(\mathcal{X}) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$, say $\delta = \langle s_1, s_2, \ldots, s_n \rangle$, $s_i = \langle r_{i,1}, r_{i,2}, \ldots, r_{i,m_i} \rangle$, where*

- *For each $1 \leq i < m_1$, $(r_{1,i}, r_{1,i+1}) \in E_{\mathsf{call}}$; and*
- *r_{1,m_1} is an* enter *node; and*
- *For each $1 \leq i < n$, s_{i+1} is a successor of s_i.*

□

Intuitively, an execution path is a sequence of execution stacks that corresponds to a possible execution of the program that emitted the executions $\mathcal{X}$. Notice that it only requires the sequence of execution stacks to be allowed by the execution graph (captured

by the notion of successor, which is defined in Definition 3.1.6), which might or might not have appeared in the executions $\mathcal{X}$ from which the execution graph $\textsc{eg}(\mathcal{X})$ is built.

Definition 3.1.8 (Language accepted by $\textsc{eg}(\mathcal{X})$) *The* language accepted by $\textsc{eg}(\mathcal{X})$*, denoted* $L_{\mathrm{EG}(\mathcal{X})}$*, is the set of all execution paths in* $\textsc{eg}(\mathcal{X})$*.* □

Each string in the language accepted by an execution graph is a sequence of execution stacks. Each execution stack consists of a sequence of integers, which intuitively represents a system call and the return addresses of the active functions when the system call is made. Though we have defined execution graphs built from observations including return addresses, they also have a black-box variant: In the case where only the system call number is used to describe a system call (return addresses are not extracted), an execution stack consists of only one integer, which is the system call number. Consequently a string in the language will become a sequence of system call numbers. We do not discuss this variation further.

3.2 Control Flow Graphs

We briefly stated in Chapter 1 that the goal of our technique is to build a model that accepts system call sequences that would be accepted by a model built from the control flow graph of the program. In this section, we define control flow graphs and the language a control flow graph accepts.

A control flow graph is an abstract representation of a procedure or program. In this thesis, it is convenient to consider a variation on the traditional control flow graph for a program P, denoted $\textsc{cfg}(P)$. First, $\textsc{cfg}(P)$ consists of a number of *control flow subgraphs*, one per function F in P, denoted $\textsc{cfsg}(F)$. Second, since we are interested only in function calls and system calls in P, each $\textsc{cfsg}(F)$ has one node per function call and two nodes per system call that it contains, in addition to its entry and exit node, and no other nodes.

Though these variations render CFG(P) different from a traditional control flow graph, we will still refer to it as one.

In this section, we refer to a *jump* as a nonsequential transfer of control, distinct from a function call or a system call. With this, we define the relationship between two instructions in a function.

Definition 3.2.1 (Follow) *Instruction t'* follows *instruction t iff t and t' are in the same function and*

- **(Base case)** *t' is at a higher address than t, and there is no jump, function call or system call between t and t';*
- **(Induction)** *There exists a jump c and a corresponding jump target c', such that t'* follows *c' and c* follows *t.*

□

The above definition defines the relative position of two instructions in a function. Next we define control flow subgraph (CFSG) and call nodes in a CFSG. In order to simplify the definition, we assuming that there are two no-op instructions in each function F denoting the starting and ending of F respectively.

Definition 3.2.2 (Control flow subgraph, call node) *A* control flow subgraph *for a function F is a directed graph* CFSG(F) $= (V, E)$*. V contains*

- *A* function call node *per function call in F;*
- *A* system call node *per system call in F;*
- *A* system call number node *per system call in F;*
- *A designated F.*enter *node and a designated F.*exit *node.*

Function call nodes and system call nodes are the call nodes of $\text{CFSG}(F)$. *Each node is identified by a label.* $(u, v) \in E$ *iff*

- *The instruction that corresponds to* v *follows (as defined in Definition 3.2.1) the instruction that corresponds to* u*; or*
- u *is a* system call node *and* v *is the corresponding* system call number node.

□

Each node in a CFSG has a label. The label of a call node could be assigned as the address of the instruction that immediately follows the call if static analysis is applied on binaries, as assumed in Section 3.3.2 for convenience. The label of a system call number node is the corresponding system call number. As in the definition of execution graphs, we do not differentiate a node and its label, i.e., in the above definition, u and v denote both the nodes and their labels.

The control flow graph $\text{CFG}(P)$ of a program P is obtained by connecting control flow subgraphs of each function in P together to form a new graph.

Definition 3.2.3 (Control flow graph) *Let* P *be a program consisting of functions* $F_1, F_2, \ldots, F_n$. *Let* $\text{CFSG}(F_i) = (V_i, E_i)$ *denote the control flow subgraph for* F_i. *The* control flow graph *for* P *is a directed graph* $\text{CFG}(P) = (V, E)$, *where* $V = \bigcup_i V_i$ *and where* $(u, v) \in E$ *iff*

- *For some* $1 \leq i \leq n$, $(u, v) \in E_i$*; or*
- $v = F_i.\mathsf{enter}$ *and* u *is a function call node representing a call to* F_i*; or*
- $u = F_i.\mathsf{exit}$ *and* v *is is a function call node representing a call to* F_i. □

Figure 3.2 shows the control flow graph of the program in Example 3.1.1 (the source code is shown in Figure 3.1).

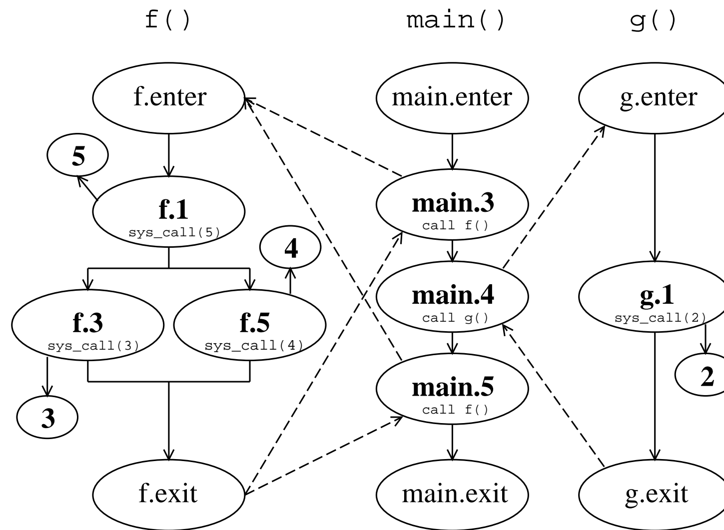


Figure 3.2: Control flow graph of Example 3.1.1

A control flow graph as described above defines all possible executions of a program P, in terms of the function and system calls it makes. During program execution, nodes in the control flow graph are traversed by following the directed edges. An execution of the program can be described by a path through which the nodes are traversed. A *call cycle* corresponds to the calling and returning of a function.

Definition 3.2.4 (Call cycle) *A sequence of nodes* $\langle v_1, v_2, \ldots, v_n \rangle$ *in* $\text{CFG}(P) = (V, E)$ *is a* call cycle *iff for some function* $F \in P$ *and the corresponding* $\text{CFSG}(F) = (V_F, E_F)$ *in* $\text{CFG}(P)$,

- **(Base case)**
  - $v_1 = v_n$ *is a function call node representing a call to* F*,* $v_2 = F.\mathsf{enter}$*,* $v_{n-1} = F.\mathsf{exit}$*; and*
  - *For each* $1 < i < n-1$*,* $v_i \in V_F$*; and*
  - *For each* $1 \leq i < n$*,* $(v_i, v_{i+1}) \in E$*.*

- **(Induction)** *For some integers* k *and* k'*, where* $1 < k < k' < n - 1$*,*
  - $v_1 = v_n$ *is a function call node representing a call to* F*,* $v_2 = F.\mathsf{enter}$*,* $v_{n-1} = F.\mathsf{exit}$*; and*
  - *For each* $1 \leq i < n$*,* $(v_i, v_{i+1}) \in E$*; and*
  - *For each* $1 < i \leq k$ *and* $k' \leq i < n - 1$*,* $v_i \in V_F$*; and*
  - $\langle v_k, v_{k+1}, \ldots, v_{k'} \rangle$ *is a call cycle.*

□

A *series of call cycles* is a concatenation of at least one call cycle.

Definition 3.2.5 (Observable path) *An* observable path π *in* CFG$(P) = (V, E)$ *is a sequence of nodes, say* $\langle v_1, v_2, \ldots, v_n \rangle$*, where*

- $v_1 =$ `main`.enter*, i.e., the entry node for the first function called in the program; and*
- v_n *is a system call node; and*
- *For each* $1 \leq i < n$*,* $(v_i, v_{i+1}) \in E$ *; and*
- *For each* $1 < i < n$*, if* v_i *is a function call node representing a call to* F*, then either* $v_{i+1} = F.\mathsf{enter}$ *or* $v_{i-1} = F.\mathsf{exit}$*; if* $v_{i-1} = F.\mathsf{exit}$*, then there exists an integer* i'*, where* $1 < i' < i$*, such that* $\langle v_{i'}, v_{i'+1}, \ldots, v_i \rangle$ *is a call cycle.*

□

The path defined in Definition 3.2.5 is called *observable* because it induces a system call, and thus intuitively would be visible to an intrusion detection system monitoring system calls. Numerous white-box process monitors additionally keep track of the active function calls in the process running the program, based on information gathered from static analysis of the program. We define active calls on an observable path as follows.

Definition 3.2.6 (Active calls on an observable path) *Let* $\pi = \langle v_1, v_2, \ldots, v_n \rangle$ *be an observable path in* $\textsc{cfg}(P) = (V, E)$. *We define the sequence of* active calls *on* π*, denoted* $A(\pi)$*, to be the result of the following procedure.*

1. *Delete all call cycles on* π*;*

2. *Denote the remaining nodes by* $\langle v_{i_1}, v_{i_2}, \ldots, v_{i_k} \rangle$*, where for each* $1 \leq j < k$*,* $i_j < i_{j+1}$*. For each* $1 \leq j < k$*, delete* v_{i_j} *unless* v_{i_j} *is a function call node;*

3. *Append a node* v_{n+1} *to the end of the sequence, where* v_{n+1} *is the system call number node such that* $(v_n, v_{n+1}) \in E$*.*

□

Since v_n (the last node on an observable path) does not belong to any call cycles, it is not deleted in the first step of the procedure in Definition 3.2.6. As such, $v_{i_k} = v_n$ in the second step of the procedure in Definition 3.2.6, and this node is not deleted in the second step either (since only nodes v_{i_j} for $1 \leq j < k$ are eligible to be deleted). In other words, v_n is always the second last element in the output of $A(\pi)$, with the last element being the system call number.

Definition 3.2.7 (Language accepted by $\textsc{cfg}(P)$**)** *Let* Π *be the set of all observable paths in* $\textsc{cfg}(P)$*, and for any* $\pi \in \Pi$*, let* $\mathsf{pre}(\pi) = \langle \pi_1, \pi_2, \ldots, \pi_n \rangle$ *denote all the observable prefixes of* π *in order of increasing length, where* $\pi_n = \pi$*. Then, the* language accepted by $\textsc{cfg}(P)$ *is*

$$L_{\mathrm{CFG}(P)} \quad = \quad \{\langle A(\pi_1), \ldots, A(\pi_n) \rangle : [\exists \pi \in \Pi : \mathsf{pre}(\pi) = \langle \pi_1, \ldots, \pi_n \rangle]\}$$

□

Notice that we define the language accepted by CFG(P) in terms of the system calls it makes and the active functions when each system call is made. A string in the language is a sequence of symbols, each of which describes a system call made by the program.

Example 3.2.1 *Figure 3.3 shows the source code and the control flow graph of a very simple program, which consists of four functions and makes four different system calls. In the program shown in Figure 3.3, the second system call made is* `read`*, which corresponds to the system call number* `3`*. The following is an observable path from* main.enter *to the node that makes this system call.*

$$\pi_1 = \langle \mathsf{main.enter}, \mathtt{main.1}, \mathtt{main.2}, \mathtt{f}.\mathsf{enter}, \mathtt{f.1}, \mathsf{g.enter}, \ldots, \mathsf{g.exit}, \mathtt{f.1}, \mathtt{f.2} \rangle$$

When trying to find the active calls on π_1, `f.1`*,* g.enter*, ...,* g.exit*,* `f.1` *should be deleted in the first step of the procedure in Definition 3.2.6, since they correspond to a call cycle (a completed function call).* main.enter *and* `main.1` *should be deleted in the second step of the procedure in Definition 3.2.6, as they are not function call nodes. Therefore,*

$$A(\pi_1) = \langle \mathtt{main.2}, \mathtt{f.2}, 3 \rangle$$

In this example, the language accepted by the control flow graph is

$$L_{\mathrm{CFG}(Ex[3.2.1])} = \{\langle \mathtt{main.1}, 5 \rangle, \langle \mathtt{main.2}, \mathtt{f.2}, 3 \rangle, \langle \mathtt{main.2}, \mathtt{f.3}, \mathtt{h.1}, 4 \rangle, \langle \mathtt{main.2}, \mathtt{f.3}, \mathtt{h.2}, 6 \rangle\}$$

Figure 3.4 shows an execution graph built from executions of the program in Example 3.2.1, assuming the input covers all possible paths of the program.[3]

□

[3] Nodes in an execution graph are denoted by integers only. In Figure 3.4 we show the correspondence with nodes in the control flow graph of the program, i.e., the line number and function name, for the purpose of illustration.

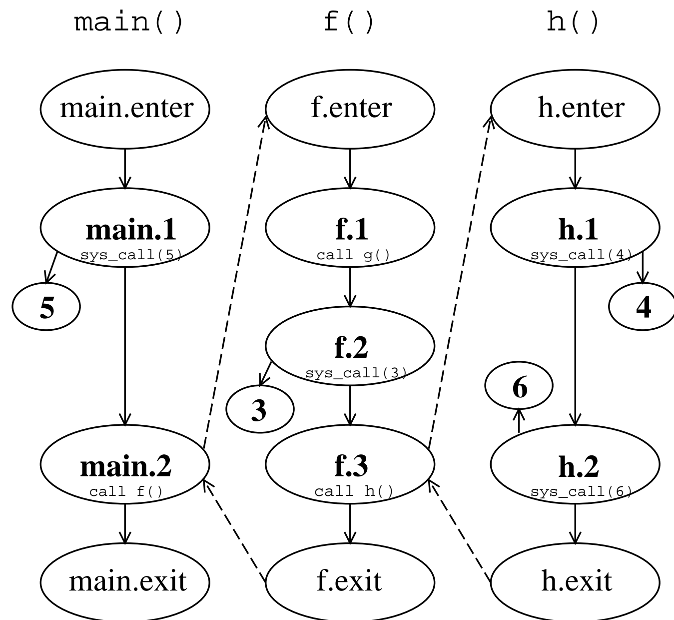
```
    int main(int argc,
      char *argv[]) {
1:    sys_call(5);
2:    f();
    }

    void f() {
1:    g();
2:    sys_call(3);
3:    h();
    }

    void h() {
1:    sys_call(4);
2:    sys_call(6);
    }

    void g() {
      ...
    }
```

(a) source code of Example 3.2.1



(b) control flow graph of Example 3.2.1 (CFSG(g()) is not shown)

Figure 3.3: Source code and control flow graph of Example 3.2.1

Notice that the languages accepted by the execution graphs of the two examples (Example 3.1.1 and Example 3.2.1) are very similar to the languages accepted by their control flow graphs. (In particular the only differences are the values of the labels.) Intuitively this similarity is what we are trying to achieve and why execution graphs are very useful in anomaly detection. In Section 3.3.2 we will formally introduce the relationship between the two, by showing two very useful properties of execution graphs.

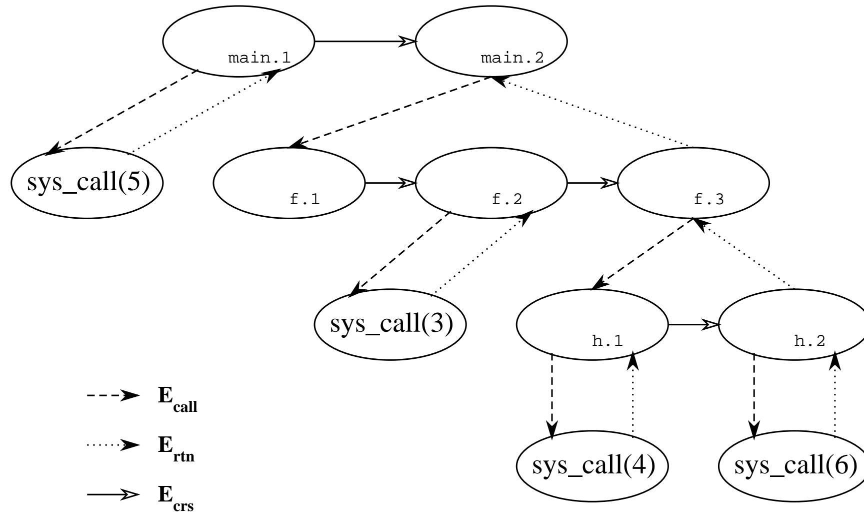


Figure 3.4: Execution graph of Example 3.2.1

3.3 Properties of Execution Graphs

In this section, we formalize two important properties of an execution graph. First, it accepts only system call sequences that are consistent with the control flow graph of the program. Second, it is maximal given a set of training data, meaning that any extensions to an execution graph could permit some intrusions to go undetected.

To this point in the chapter, we have not specified the program executions that are *useful* to build an execution graph (though any execution results in one). However, to prove a relationship with the control flow graph of the program, it is necessary to specify which executions are useful for this purpose. Intuitively, these executions are ones that do not include an attack, and more specifically, for which the return addresses are a reliable reflection of the intended execution of the underlying program. We refer to such executions as *well-behaved*.

3.3.1 Well-behaved executions

More precisely, denote the execution of program P on input I by $P(I)$. Input string I includes all inputs to the process running P since its initialization, and can include multiple "invocations" if program P is a server program. In this case, the multiple invocations of P are separated in I in a canonical way. The runtime process that executes $P(I)$ maintains a call stack in conformance with certain conventions, induced via the function call and return code emitted by the compiler for the language. While we do not detail these conventions here, we expect that the return address of each stack frame is inserted when the function call occurs and is not modified until return from the function—at which time the stack frame is destroyed. We say that a program P is "well-behaved" on an input I if the execution $P(I)$ conforms strictly to this expectation, i.e., that return address fields in stack frames are modified *only* in this fashion, and the stack frames are created only when function calls are made by the program P.

Definition 3.3.1 (Well-behaved executions) *Program P is* well-behaved *on input I if execution $P(I)$ maintains a call stack consisting of stack frames, one per active function call, and such that the return address in each stack frame is not modified while the corresponding function call is active.* □

Of course, a common method of exploiting a vulnerable program P involves running P on an input I' for which it is not well-behaved, i.e., that modifies a return address on the stack when the function call is still active.

The anomaly detector that we describe in this section is assumed to be trained on the observed behaviors (emitted system calls) in executions $P(I_1), \ldots, P(I_k)$ where P is well-behaved on each I_j. In this way, the return addresses extracted from the stack (as in [FKF+03]) reflect the execution of the program. We denote these executions $P(\mathcal{I}) = \{P(I_1), \ldots, P(I_k)\}$.

3.3.2 Properties of Execution Graphs

Recall that an execution graph is a model constructed by a gray-box technique. None of the previous gray-box techniques, to our knowledge, has been formally related to the control flow graph of the underlying program. The execution graph differs from these approaches in the sense that the language accepted by an execution graph can be directly related to the language accepted by the control flow graph of the underlying program. Moreover, this relationship can be proved analytically. This is a significant improvement since goals of many white-box techniques can now be achieved using gray-box techniques, i.e., without static analysis on the source code or binary.

Here we show two theorems of the execution graph and the control flow graph of a program. Without loss of generality, we assume that the label of a call node in the control flow graph is the address of the instruction that immediately follows the function call or system call, which is easily obtained by static analysis of the binary. If this is not the case, e.g., if static analysis is applied on the source code, there is always a one-to-one mapping between the labels and these addresses. For convenience, we omit this mapping in the following theorems.

Theorem 3.3.1 *If P is a program that is well-behaved on input $\mathcal{I}$, then $L_{\mathrm{EG}(P(\mathcal{I}))} \subseteq L_{\mathrm{CFG}(P)}$.*

We first prove the following lemmas. As stated in Section 3.3.2, without loss of generality, we assume that the label of any call node in the control flow graph is in fact the address of the instruction that immediately follows the call. If this is not the case, e.g., if the control flow graph is obtained by static analysis of the source code, there is always a one-to-one mapping between the labels and these addresses. For convenience, we omit this mapping in the following proofs.

In the following proofs, we use μ to denote the length of a function call or system call

instruction. Since we assume that the label of any call node x in the control flow graph is the address of the instruction that immediately follows the call, $x - \mu$ represents the address of the corresponding call instruction. We use F_v to denote the function in P that consists of node v.

Lemma 3.3.1 *Let P be a program that is well-behaved on input $\mathcal{I}$. Let* $\textsc{eg}(P(\mathcal{I})) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$ *and* $\textsc{cfg}(P) = (V', E')$*, then* $V \subseteq V'$.

Proof.

$v \in V \wedge v$ is a leaf node

$\Rightarrow$ P is able to make a system call with system call number v

$\Rightarrow$ $v \in V'$

$v \in V \wedge v$ is not a leaf node

$\Rightarrow$ v is one of the return addresses observed when P makes a system call

$\Rightarrow$ $(v - \mu)$ is the address of a call instruction □

$\Rightarrow$ $(v - \mu)$ corresponds to some function or system call site in P

$\Rightarrow$ $v \in V'$

Notice that there could be $v' \notin V$ while $v' \in V'$, because input $\mathcal{I}$ does not necessarily cover all possible executions of P, and that some executions allowed by $\textsc{cfg}(P)$ might never appear in actual runs.

Lemma 3.3.2 *Let P be a program that is well-behaved on input $\mathcal{I}$. If* $\langle r_1, r_2, \ldots, r_l, \ldots\rangle$ *and* $\langle r'_1, r'_2, \ldots, r'_l, \ldots\rangle$ *are two observations in* $P(\mathcal{I})$*, such that for each* $1 \leq i < l$, $r_i = r'_i$, $r_l \neq r'_l$*, then for some function* $F \in P$, r_l *and* r'_l *are both in* $\textsc{cfsg}(F)$.

Proof.

$\langle r_1, r_2, \ldots, r_l, \ldots\rangle$ and $\langle r'_1, r'_2, \ldots, r'_l, \ldots\rangle$ are two observations

$\Rightarrow$ $(r_l - \mu)$ is in a function that is called from $(r_{l-1} - \mu)$; $(r'_l - \mu)$ is in a function that is called from $(r'_{l-1} - \mu)$

Now, for each $1 \leq i < l$, $r_i = r'_i$, $r_l \neq r'_l$

$\Rightarrow$ $(r_{l-1} - \mu) = (r'_{l-1} - \mu)$

$\Rightarrow$ $(r_l - \mu)$ and $(r'_l - \mu)$ are in the same function (instruction at address $(r_{l-1} - \mu)$ can call only one function) $\square$

$\Rightarrow$ r_l and r'_l are nodes in the same CFSG

Lemma 3.3.3 *Let P be a program that is well-behaved on input $\mathcal{I}$. Let* EG$(P(\mathcal{I})) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$ *and* CFG$(P) = (V', E')$. *If $(r, r') \in E_{\mathsf{crs}}$, then there exist a sequence of nodes $\langle v_1, v_2, \ldots, v_n \rangle$ in* CFG(P) *such that*

- *$v_1 = r$, $v_n = r'$; and*
- *For each $1 \leq i < n$, $(v_i, v_{i+1}) \in E'$; and*
- *For each $1 < i < n$, v_i is not a system call node; and*
- *If $n > 2$, then $\langle v_2, v_3, \ldots, v_{n-1} \rangle$ is a (series of) call cycle(s).*

Proof.

$(r, r') \in E_{\mathsf{crs}}$

$\Rightarrow$ there exists two consecutive observations o and o' in $P(\mathcal{I})$, where $o = \langle r_1, r_2, \ldots, r_l, \ldots r_n \rangle$, $o' = \langle r'_1, r'_2, \ldots, r'_l, \ldots r'_{n'} \rangle$, such that for each $1 \leq i < l$, $r_i = r'_i$, and $r_l = r$ and $r'_l = r'$ (Definition 3.1.3)

o and o' are two consecutive observations

$\Rightarrow$ there must be a path in CFG(P) from r_{n-1} to $r'_{n'-1}$ via r_l and r'_l that does not consist of any other system call nodes

For each $1 \leq i < l$, $r_i = r'_i$, $r_l \neq r'_l$

$\Rightarrow$ r_l and r'_l are addresses in the same function (Lemma 3.3.2)

$\Rightarrow$ any function call nodes on the path from r to r' must form a (series of) call cycles (completed function calls) $\square$

$\Rightarrow$ there must be a path in CFG(P) from r to r' that satisfies the claimed properties.

Lemma 3.3.4 *Let P be a program that is well-behaved on input $\mathcal{I}$. Let* EG$(P(\mathcal{I})) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$ *and* CFG$(P) = (V', E')$. *If $(r, r') \in E_{\mathsf{call}}$ and r' is not a leaf node, then there exists a sequence of nodes $\langle v_1, v_2, \ldots, v_n \rangle$ in* CFG(P) *such that*

- $v_1 = r$, $v_2 = F_{r'}.\mathsf{enter}$, $v_n = r'$; *and*
- *For each* $1 \leq i < n$, $(v_i, v_{i+1}) \in E'$; *and*
- *For each* $3 \leq i < n$, v_i *is not a system call node; and*
- *If* $n > 3$, *then* $\langle v_3, v_4, \ldots, v_{n-1} \rangle$ *is a (series of) call cycle(s).*

Proof.

According to Definition 3.1.3, $(r, r') \in E_{\mathsf{call}}$ results from at least one of the following three conditions. We prove Lemma 3.3.4 in all these three conditions.

- (Base case of Definition 3.1.3)

There exists two consecutive observations o and o' in $P(\mathcal{I})$, where $o = \langle r_1, r_2, \ldots, r_l, \ldots \rangle$, $o' = \langle r'_1, r'_2, \ldots, r'_l, \ldots, r, r', \ldots \rangle$, and for each $1 \leq i < l$, $r_i = r'_i$ and $(r_l, r'_l) \in E_{\mathsf{crs}}$

$\Rightarrow$ after the system call corresponding to o is executed, execution has to return to CFSG(F_{r_l}) and then follow the path as described in Lemma 3.3.3 and subsequently enter CFSG(F_r) and CFSG($F_{r'}$) in order to make system call that corresponds to o'

$\Rightarrow$ instruction at $(r - \mu)$ calls function $F_{r'}$, and there must be a path in CFG(P) from $F_{r'}.\mathsf{enter}$ to r' that satisfies the claimed properties.

- (First induction of Definition 3.1.3) Given $(x_0, x_1) \in E_{\mathsf{call}}$, $x_1 \stackrel{\mathsf{crs}}{\rightarrow} x_2$, $(x_2, x_3) \in E_{\mathsf{rtn}}$, $x_3 = r$ and $x_1 = r'$

  $(x_0, x_1) \in E_{\mathsf{call}}$

  $\Rightarrow$ there exists a path in CFG(P) from $F_{x_1}.\mathsf{enter}$ to x_1 that satisfies the claimed properties. (Base case in this proof)

  Since we have already found the path from $F_{x_1}.\mathsf{enter}$ to x_1 that satisfies the claimed properties, it only remains to prove that $(x_3, F_{x_1}.\mathsf{enter}) \in E'$.

  $x_1 \stackrel{\mathsf{crs}}{\rightarrow} x_2$

  $\Rightarrow$ $F_{x_1} = F_{x_2}$ (Lemma 3.3.2)

  $(x_2, x_3) \in E_{\mathsf{rtn}}$

  $\Rightarrow$ $(F_{x_2}.\mathsf{exit}, x_3) \in E'$

  $\Rightarrow$ $(x_3, F_{x_2}.\mathsf{enter}) \in E'$

  $\Rightarrow$ $(x_3, F_{x_1}.\mathsf{enter}) \in E'$

- (Second induction of Definition 3.1.3) Given $(x_0, x_1) \in E_{\mathsf{call}}$, $x_1 \stackrel{\mathsf{crs}}{\rightarrow} x_2$, and $(x_3, x_2) \in E_{\mathsf{call}}$,

- When $x_3 = r$ and $x_1 = r'$

$$(x_0, x_1) \in E_{\mathsf{call}}$$

$\Rightarrow$ there exists a path in CFG(P) from $F_{x_1}.\mathsf{enter}$ to x_1 that satisfies the claimed properties. (Base case in this proof)

Since we have already found the path from $F_{x_1}.\mathsf{enter}$ to x_1 satisfying the claimed properties, it only remains to prove that $(x_3,\ F_{x_1}.\mathsf{enter}) \in E'$.

$$x_1 \stackrel{\mathsf{crs}}{\rightarrow} x_2$$

$\Rightarrow$ $F_{x_1} = F_{x_2}$ (Lemma 3.3.2)

$$(x_3, x_2) \in E_{\mathsf{call}}$$

$\Rightarrow$ $(x_3, F_{x_2}.\mathsf{enter}) \in E'$

$\Rightarrow$ $(x_3, F_{x_1}.\mathsf{enter}) \in E'$

- When $x_0 = r$ and $x_2 = r'$

$$(x_3, x_2) \in E_{\mathsf{call}}$$

$\Rightarrow$ there exists a path in CFG(P) from $F_{x_2}.\mathsf{enter}$ to x_2 that satisfies the claimed properties. (Base case in this proof)

Since we have already found the path from $F_{x_2}.\mathsf{enter}$ to x_2 satisfying the claimed properties, it only remains to prove that $(x_0,\ F_{x_2}.\mathsf{enter}) \in E'$.

$$x_1 \stackrel{\mathsf{crs}}{\rightarrow} x_2$$

$\Rightarrow$ $F_{x_1} = F_{x_2}$ (Lemma 3.3.2)

$$(x_0, x_1) \in E_{\mathsf{call}}$$

$\Rightarrow$ $(x_0, F_{x_1}.\mathsf{enter}) \in E'$

$\Rightarrow$ $(x_0, F_{x_2}.\mathsf{enter}) \in E'$

□

Analogous to Lemma 3.3.4, we have Lemma 3.3.5 as shown below (proof of which is skipped).

Lemma 3.3.5 *Let P be a program that is well-behaved on input $\mathcal{I}$. Let* $\textsc{eg}(P(\mathcal{I})) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$ *and* $\textsc{cfg}(P) = (V', E')$. *If* $(r, r') \in E_{\mathsf{rtn}}$ *and r is not a leaf node, then there exists a sequence of nodes* $\langle v_1, v_2, \ldots, v_n \rangle$ *in* $\textsc{cfg}(P)$ *such that*

- $v_1 = r$, $v_{n-1} = F_r.\mathsf{exit}$, $v_n = r'$; *and*
- *For each* $1 \leq i < n$, $(v_i, v_{i+1}) \in E'$; *and*
- *For each* $1 < i \leq n-3$, v_i *is not a system call node; and*
- *If* $n > 3$, *then* $\langle v_2, v_3, \ldots, v_{n-2} \rangle$ *is a (series of) call cycle(s).*

Lemma 3.3.6 *Let P be a program that is well-behaved on input $\mathcal{I}$. Let* $\textsc{eg}(P(\mathcal{I})) = (V, E_{\mathsf{call}}, E_{\mathsf{crs}}, E_{\mathsf{rtn}})$. *If* $\langle r_1, r_2, \ldots, r_n \rangle$ *is an execution stack, then there exists an observable path π in* $\textsc{cfg}(P)$ *such that* $A(\pi) = \langle r_1,\ r_2,\ \ldots,\ r_n \rangle$.

Proof.

$\langle r_1, r_2, \ldots, r_n \rangle$ is an execution stack

$\Rightarrow$ for each $1 \leq i < n$, $r_i \stackrel{\mathsf{xcall}}{\rightarrow} r_{i+1}$ (Definition 3.1.5)

$\Rightarrow$ for each $1 \leq i < n$, $(r_i, r_{i+1}) \in E_{\mathsf{call}}$ or $\left(\exists z : (r_i, z) \in E_{\mathsf{call}} \wedge z \stackrel{\mathsf{crs}}{\rightarrow} r_{i+1}\right)$ (Definition 3.1.5)

If for any $1 \leq i < n-1$, $(r_i, r_{i+1}) \in E_{\mathsf{call}}$

$\Rightarrow$ there exists a path $\langle r_i, F_{r_{i+1}}.\mathsf{enter},\ \ldots,\ r_{i+1} \rangle$ in $\textsc{cfg}(P)$ (Lemma 3.3.4).

If for any $1 \leq i < n-1$, $(r_i, z) \in E_{\mathsf{call}} \wedge z \stackrel{\mathsf{crs}}{\rightarrow} r_{i+1}$

$\Rightarrow$ there exists a path $\langle r_i,\ F_{r_{i+1}}.\mathsf{enter},\ \ldots,\ z,\ \ldots,\ r_{i+1} \rangle$ in $\textsc{cfg}(P)$ (Lemma 3.3.4 and Lemma 3.3.3).

Connecting $\langle \mathtt{main}.\mathsf{enter}, \ldots, r_1 \rangle$ and all these paths from each r_i to r_{i+1} together forms an observable path π that traverses $r_1, r_2, \ldots, r_{n-1}$.

Since each individual path $\langle r_i, \ldots, r_{i+1}\rangle$ consists of only $r_i, F_{r_{i+1}}.\textsf{enter}, r_{i+1}$ and a (possibly empty series of) call cycle(s) (Lemma 3.3.4), $A(\pi) = \langle r_1, r_2, \ldots, r_n\rangle$ (Definition 3.2.6). □

Lemma 3.3.7 *Let P be a program that is well-behaved on input $\mathcal{I}$. Let* $\textsc{eg}(P(\mathcal{I})) = (V, E_{\textsf{call}}, E_{\textsf{crs}}, E_{\textsf{rtn}})$ *and* $\textsc{cfg}(P) = (V', E')$. *If* $s = \langle r_1, r_2, \ldots, r_m\rangle$ *and* $s' = \langle r'_1, r'_2, \ldots, r'_{m'}\rangle$ *are execution stacks in V, and s' is a successor of s, then there exist a sequence of nodes* $\langle v_1, v_2, \ldots, v_n\rangle$ *in* $\textsc{cfg}(P)$ *such that*

- $v_1 = r_{m-1}$, $v_n = r'_{m'-1}$; *and*
- *For each* $1 \le i < n$, $(v_i, v_{i+1}) \in E'$; *and*
- *For each* $1 < i < n$, v_i *is not a system call node; and*
- *Let* $\langle v_{l_1}, v_{l_2}, \ldots, v_{l_j}\rangle$ *denote the remaining sequence of nodes when all call cycles on* $\langle v_1, v_2, \ldots, v_n\rangle$ *are removed, where for each* $1 \le i < j$, $l_i < l_{i+1}$. *Then there exists an integer k, such that*
  - *For each* $1 \le i < k$, $r_i = r'_i$; *and*
  - $j = 2(m + m' - 2k - 1)$; *and*
  - $v_{l_1} = r_{m-1}$, $v_{l_2} = F_{r_{m-1}}.\textsf{exit}$;

    $\ldots$;

    $v_{l_{2(m-k)-3}} = r_{k+1}$, $v_{l_{2(m-k)-2}} = F_{r_{k+1}}.\textsf{exit}$;

    $v_{l_{2(m-k)-1}} = r_k$, $v_{l_{2(m-k)}} = r'_k$;

    $v_{l_{2(m-k)+1}} = F_{r'_{k+1}}.\textsf{enter}$, $v_{l_{2(m-k)+2}} = r'_{k+1}$;

    $\ldots$;

    $v_{l_{2(m+m'-2k)-3}} = F_{r'_{m'-1}}.\textsf{enter}$, $v_{l_{2(m+m'-2k)-2}} = r'_{m'-1}$.

Proof.

s' is a successor of s

$\Rightarrow$ there exists an integer k such that $r_m \stackrel{\mathsf{rtn}}{\to} r_k$, $(r_k, r'_k) \in E_{\mathsf{crs}}$, $r'_k \stackrel{\mathsf{call}}{\to} r'_{m'}$ and for each $1 \le i < k$, $r_i = r'_i$ (Definition 3.1.6)

$\Rightarrow$ there exist three paths from r_{m-1} to r_k (Lemma 3.3.5), from r_k to r'_k (Lemma 3.3.3) and from r'_k to $r'_{m'-1}$ (Lemma 3.3.4)

$\Rightarrow$ connecting the above 3 paths together forms the sequence of nodes with the claimed properties. □

Lemma 3.3.8 *Let P be a program that is well-behaved on input $\mathcal{I}$. If there is an execution path $\delta = \langle s_1, \ldots, s_n \rangle$ in $\text{EG}(P(\mathcal{I}))$, where $s_i = \langle r_{i,1}, \ldots, r_{i,m_i} \rangle$, then there exists an observable path $\pi = \langle \texttt{main}.\mathsf{enter}, \ldots, r_{1,m_1-1}, \ldots, r_{2,m_2-1}, \ldots, r_{n,m_n-1} \rangle$ in $\text{CFG}(P)$ such that*

- *$r_{1,m_1-1}, r_{2,m_2-1}, \ldots, r_{n,m_n-1}$ are the only system call nodes on π; and*
- *Let $\mathsf{pre}(\pi) = \langle \pi_1, \pi_2, \ldots, \pi_n \rangle$, then for each $1 \le i \le n$, $A(\pi_i) = s_i$.*

Proof.

According to Lemma 3.3.6, there exists a path $\beta_0 = \langle \texttt{main}.\mathsf{enter}, \ldots, r_{1,1}, F_{r_{1,2}}.\mathsf{enter}, \ldots, r_{1,2}, \ldots, r_{1,m_1-1} \rangle$ in $\text{CFG}(P)$. Since for each $1 \le i < m_1$, $(r_{1,i}, r_{1,i+1}) \in E_{\mathsf{call}}$ (Definition 3.1.7), r_{1,m_1-1} is the only system call node on β_0 (Lemma 3.3.4).

According to Lemma 3.3.7, for each $1 \le i < n$, there is a path $\beta_i = \langle r_{i,m_i-1}, \ldots, F_{r_{i,m_i-1}}.\mathsf{exit}, \ldots, F_{r_{i,k_i+1}}.\mathsf{exit}, r_{i,k_i}, \ldots, r_{i+1,k_i}, F_{r_{i+1,k_i+1}}.\mathsf{enter}, \ldots, F_{r_{i+1,m_{i+1}-1}}.\mathsf{enter}, \ldots, r_{i+1,m_{i+1}-1} \rangle$, where for each $1 \le j < k_i$, $r_{i,j} = r_{i+1,j}$.

Connecting $\beta_0, \beta_1, \ldots, \beta_{n-1}$ together forms π.

Since the only system call nodes on β_i are r_{i,m_i-1} and $r_{i+1,m_{i+1}-1}$ (Lemma 3.3.7), $r_{1,m_1-1}, r_{2,m_2-1}, \ldots, r_{n,m_n-1}$ are the only system call nodes on π.

Let $k_0 = m_1 - 1$ and for each $1 \le i < n$, let $l_i = \min(k_{i-1}, k_i)$. Since the subsequence $\langle r_{i,l_i}, F_{r_{i,l_i+1}}.\mathsf{enter}, \ldots, F_{r_{i,m_i-1}}.\mathsf{enter}, \ldots, r_{i,m_i-1} \rangle$ from β_{i-1} and $\langle r_{i,m_i-1}, \ldots, F_{r_{i,m_i-1}}.\mathsf{exit},$

$\ldots$, $F_{r_{i,l_i+1}}.\mathsf{exit}$, $r_{i,l_i}\rangle$ from β_i forms a (possibly empty) call cycle, they will be deleted from $A(\pi)$ (First step of the procedure in Definition 3.2.6).

Since for each $1 \leq i < n$ and each $1 \leq j < k_i$, $r_{i,j} = r_{i+1,j}$ (Definition 3.1.6), the remaining nodes of $A(\pi)$ after the second step of the procedure in Definition 3.2.6 are $\langle r_{n,1}$, $r_{n,2}$, $\ldots$, $r_{n,m_n-1}\rangle$. Therefore, $A(\pi) = s_n$.

$A(\pi_i) = s_i$ can be proved similarly. □

Theorem 3.3.1 follows immediately from Lemma 3.3.8.

Theorem 3.3.1 says that the language accepted by an execution graph is a subset of the language accepted by the control flow graph of the program, which is a property unavailable in most other gray-box techniques. It provides another level of confidence: if some execution is allowed by an execution graph, it is guaranteed that the execution is not only normal ("similar" to past executions), but also valid (allowed by the control flow graph). Such a property could only be achieved previously by white-box techniques.

Theorem 3.3.1 only says that $L_{\mathrm{EG}(P(\mathcal{I}))} \subseteq L_{\mathrm{CFG}(P)}$. They are not equal because, e.g., the input $\mathcal{I}$ might not cover all possible executions of the program, in which case there is no way for $\mathrm{EG}(P(\mathcal{I}))$ to safely accept such a missing execution, even with the inductive definition in Definition 3.1.3.

Theorem 3.3.2 shows that if the execution graph were to be extended to allow any additional strings in the language, it could accept some intrusions that program P does not allow.

Theorem 3.3.2 *Let $\mathcal{I}$ be a set of inputs, and $\mathrm{EG}(P(\mathcal{I}))$ be an execution graph where P is well-behaved on $\mathcal{I}$. There exists a program P', which is also well-behaved on $\mathcal{I}$, such that $L_{\mathrm{CFG}(P')} = L_{\mathrm{EG}(P(\mathcal{I}))}$.*

To show the existence of such a program P', we (i) build a graph G' from the execution graph $\mathrm{EG}(P(\mathcal{I}))$; (ii) show that G' is the control flow graph of some program P' that is well-behaved on input $\mathcal{I}$, i.e., $\mathrm{CFG}(P') = G'$; and (iii) show that $L_{\mathrm{CFG}(P')} = L_{\mathrm{EG}(P(\mathcal{I}))}$.

Definition 3.3.2 **(*E2G*)** *The operation E2G takes as input an execution graph* $\textsc{eg}(P(\mathcal{I})) = (V,\ E_{\mathsf{call}},\ E_{\mathsf{crs}},\ E_{\mathsf{rtn}})$ *and performs the following operations:*

1. *For each* $x_i \in V$ *where*

   - x_i *is not a leaf node; and*
   - *there does not exist a leaf node* v *such that* $(x_i, v) \in E_{\mathsf{call}}$.

   Let

$$\begin{aligned}
C(x_i) &= \{v : (x_i, v) \in E_{\mathsf{call}}\} \\
C'(x_i) &= \left\{v : [\exists v' \in C(x_i) : v' \xrightarrow{\mathsf{crs}} v]\right\} \\
C''(x_i) &= \{v : v \text{ is a leaf node} \wedge [\exists v' \in \left(C(x_i) \cup C'(x_i)\right) : (v', v) \in E_{\mathsf{call}}]\} \\
E(x_i) &= \left\{(v_1, v_2) : v_1 \in \left(C(x_i) \cup C'(x_i)\right) \wedge v_2 \in \left(C(x_i) \cup C'(x_i)\right) \wedge (v_1, v_2) \in E_{\mathsf{crs}}\right\} \\
E'(x_i) &= \left\{(v_1, v_2) : v_1 \in \left(C(x_i) \cup C'(x_i)\right) \wedge v_2 \in C''(x_i) \wedge (v_1, v_2) \in E_{\mathsf{call}}\right\}
\end{aligned}$$

2. *Define the equivalence relation* $x_i \sim x_j$ *if* $C(x_i) = C(x_j)$*. Let* $[x_i]$ *denote the equivalence class of* x_i*. For each equivalence class* $[x_i]$*, let* $G_{[x_i]} = (V_{[x_i]},\ E_{[x_i]})$ *where*

$$\begin{aligned}
V_{[x_i]} &= C(x_i) \cup C'(x_i) \cup C''(x_i) \cup \{G_{[x_i]}.\mathsf{enter}, G_{[x_i]}.\mathsf{exit}\} \\
E_{[x_i]} &= E(x_i) \cup E'(x_i) \cup \left\{(G_{[x_i]}.\mathsf{enter}, v) : v \in C(x_i)\right\} \cup \\
&\quad \left\{(v, G_{[x_i]}.\mathsf{exit}) : v \in V_{[x_i]} \wedge (v, x_i) \in E_{\mathsf{rtn}}\right\}
\end{aligned}$$

3. Create a new graph $G' = (V', E')$, such that

$$V' = \left(\bigcup_{[x_i]} V_{[x_i]}\right) \cup M$$

$$E' = \bigcup_{[x_i]} \left(E_{[x_i]} \cup \{(v, G_{[x_i]}.\mathsf{enter}) : v \in [x_i]\} \cup \{(G_{[x_i]}.\mathsf{exit}, v) : v \in [x_i]\}\right) \cup \{(v_1, v_2) : \{v_1, v_2\} \subseteq M \wedge (v_1, v_2) \in E_{\mathsf{crs}}\}$$

where

$$M = \{v : v \in V \wedge [\textit{there does not exist } v' \in V : v' \stackrel{\mathsf{xcall}}{\to} v]\}$$

Operation E2G returns the graph G'. □

In the above definition, M is the set of nodes that represent addresses in `main()`. With Definition 3.3.2, we are done with the first step in our proof. The next step is to prove that $\textsc{cfg}(P') = G'$ for some program P' that is also well-behaved on input $\mathcal{I}$.

Lemma 3.3.9 *If graph $G' = (V', E')$ is the output of operation E2G on an execution graph $\textsc{eg}(P(\mathcal{I}))$, then there exists some program P' which is well-behaved on input $\mathcal{I}$, such that $\textsc{cfg}(P') = G'$.*

Though we do not provide the proof of Lemma 3.3.9, the following is the intuition. From Definition 3.3.2, one can notice that graph G' contains a set of subgraphs, which are connected by directed edges from a function call node to the entry node of the function subgraph, and from the exit node of the function subgraph to the same function call node. Besides that, each subgraph contains function call nodes and system call nodes, as well as one entry node and one exit node. When given this graph G', programming languages such as C and C++ can be used to implement each subgraph as a function, and implement the

entire graph G' as a program P'. If implemented correctly, the implementation output P' will be well-behaved on input $\mathcal{I}$, and the control flow graph of P' will be the same as G'.

The last step in our proof of Theorem 3.3.2 is to show that $L_{\text{CFG}(P')} = L_{\text{EG}(P(\mathcal{I}))}$, where $\text{CFG}(P') = G' = (V', E')$. To prove this we need to show that (i) $L_{\text{EG}(P(\mathcal{I}))} \subseteq L_{\text{CFG}(P')}$, and (ii) $L_{\text{CFG}(P')} \subseteq L_{\text{EG}(P(\mathcal{I}))}$. The proof of (i) is very similar to the proof of Theorem 3.3.1 and it is skipped in this thesis. We only show the important lemmas for the proof of (ii). Notice that the difference between these two proofs and those in proving Theorem 3.3.1 is that here V' and E' are given as in Definition 3.3.2, whereas in Theorem 3.3.1 they are not given.

Lemma 3.3.10 *Let P be a program that is well-behaved on input $\mathcal{I}$, and $E2G\,(\text{EG}(P(\mathcal{I}))) = G'$. If π is an observable path in G', then there exists an execution stack s in $\text{EG}(P(\mathcal{I}))$ such that $s = A(\pi)$.*

Lemma 3.3.11 *Let P be a program that is well-behaved on input $\mathcal{I}$, and $E2G\,(\text{EG}(P(\mathcal{I}))) = G'$. Let π be an observable path in G', and $\mathsf{pre}(\pi) = \langle \pi_1, \pi_2, \ldots, \pi_n \rangle$, then $\langle A(\pi_1)$, $A(\pi_2)$, $\ldots$, $A(\pi_n)\rangle$ is an execution path in $\text{EG}(P(\mathcal{I}))$.*

Theorem 3.3.2 states that for any input $\mathcal{I}$ and the execution graph obtained on input $\mathcal{I}$, there exists a program P' which is well-behaved on $\mathcal{I}$, such that the language accepted by the control flow graph of this program is the same as the language accepted by the execution graph. This means that the execution graph is the "accurate" model of some program P'. Since there exists such a program P', if the execution graph were to be extended to accept any additional string in its language, it will allow an intrusion to the program P'. Informally, this means that the execution graph is a maximal graph given the set of input.

3.4 Performance evaluation

In this section we provide insight into the likely performance of our technique in an anomaly detection system. During program monitoring there are two tasks the anomaly detector needs to perform for each system call: (i) to walk through the stack frames and obtain all return addresses; (ii) to determine whether the current system call is allowed. We previously measured the cost of extracting program return addresses and found that for a Linux kernel compilation it adds less than 6% to the overall execution time [GRS04b]. Therefore, extracting return addresses from the running process should introduce only moderate overhead.

Second, we measure the time it takes to process system calls when using our execution graph model. We observe the executions of four common FTP and HTTP server programs, `wu-ftpd`, `proftpd`, `Apache httpd`, and `Apache httpd` with a `chroot` patch, and extract the execution graphs from them. Information, including return addresses, of every system call is recorded into log files, and subsequently processed to detect anomalies. We run an experiment to measure the time it takes to process these system calls by running the anomaly detector on a desktop computer with an Intel Pentium IV 2.2 GHz CPU. The experiment was repeated for a few times. Results are virtually identical in all runs of the experiment, and the average results are shown in Table 3.1.

	wu-ftpd	proftpd	Apache	Apache with `chroot` patch
number of syscalls processed	4202602	9062102	5142088	4693300
average processing time per syscall	0.130 μs	1.063 μs	0.417 μs	0.624 μs

Table 3.1: Performance overhead for processing system calls

Although the average processing time per system call is very different for these four programs (due to the different number of functions in the program and consequently the different number of return addresses to be processed for each system call), results show that

program monitoring is extremely efficient when using the execution graph model.

Chapter 4

Behavioral Distance

Constructing a model that describes the normal behavior of a program for accurate intrusion detection is a challenging problem, especially because of *mimicry attacks* [TMK02, WS02, KKM+05, GJM06] which are able to 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 host-based anomaly detector cannot differentiate execution of the attack code from execution of the original server program. Output voting in a replicated system that detects [SR87, BB93, AMPR01] or masks [Lam78, Sch90, Rei94, CL02, CP02, YMV+03, AEMGG+05] Byzantine faults or intrusions by comparing server outputs cannot detect such attacks either. A replicated system that employs only output voting will consequently allow a compromised server that generates the correct output to leak sensitive data or to attack other machines in the network.

Behavioral distance measures the extent to which two processes behave differently when executing on the same inputs [GRS05, GRS06]. A typical use of behavioral distance is to detect carefully crafted attacks that intend to evade detection by a host-based anomaly detector, e.g., mimicry attacks [TMK02, WS02, KKM+05, GJM06], by comparing the behaviors of two diverse processes when they are executing the same, potentially malicious,

inputs. Assuming that the two processes 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.

There are many ways to monitor the "behavior" of a process. For example, one could look at sequence of instructions executed, or patterns in which process's internal states change. In this thesis, we propose a specific measure for behavioral distance, by using system call sequences emitted by processes.

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

$$\chi_1 = \langle c_{1,1}, c_{1,2}, \ldots, c_{1,l_1} \rangle \qquad \chi_2 = \langle c_{2,1}, c_{2,2}, \ldots, c_{2,l_2} \rangle \tag{4.1}$$

emitted by two processes while processing the same request. Here, each $c_{i,j}$ denotes the system call number (a natural number) of the j-th system call by the i-th process. The distance should indicate whether these sequences reflect similar activity in the processes. Producing this distance is complicated by the fact that the processes might be running on diverse platforms, and so the set of system calls $C_1 = \{c_{1,j}\}_{1 \le j \le l_1}$ on the first platform can be different from the set $C_2 = \{c_{2,j}\}_{1 \le j \le l_2}$ 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 $l_1 \neq l_2$. In order to find the correlation between the two sequences, we consider different ways of inserting dummy symbols σ into them to generate an *alignment*

$$\chi'_1 = \langle c'_{1,1}, c'_{1,2}, \ldots, c'_{1,l'_1} \rangle \qquad \chi'_2 = \langle c'_{2,1}, c'_{2,2}, \ldots, c'_{2,l'_2} \rangle \tag{4.2}$$

where $l'_1 = l'_2$.

In this thesis we propose two approaches of calculating the behavioral distance. The first approach is based on evolutionary distance (ED), and the second approach is based on a Hidden Markov Model (HMM).

4.1 System call phrases

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

System call phrases have been used in intrusion/anomaly detection systems [WDD00, GRS04b]. Working on system call phrases significantly improves the performance of behavioral distance calculation, since a relatively long system call sequence is recognized as a short sequence of system call phrases.

We use the phrase extraction algorithm TEIRESIAS [RF98] and the phrase reduction algorithm in [WDD00], which are also used in intrusion/anomaly detection systems [WDD00, GRS04b], to extract system call phrases. The TEIRESIAS algorithm analyzes system call sequences from sample executions, and outputs a set of system call phrases that are guaranteed to be maximal [RF98]. Maximal phrases (the number of occurrences of which will decrease if the phrases are extended to include any additional system call) capture system calls that are made in a fixed sequence, and therefore intuitively should conform to basic blocks/functions in the program source code. The phrase reduction algorithm takes the result from TEIRESIAS and outputs a subset of the system call phrases that are necessary

to cover the training data. Note that other phrase extraction and reduction algorithms can be used.

The set of system call phrases returned by the phrase reduction algorithm is then used to break a system call sequence into a system call phrase sequence. In our experience when dealing with normal program executions, a system call sequence usually corresponds to a unique phrase sequence. To simplify discussion, we assume that a system call sequence can be uniquely decomposed into a sequence of system call phrases in the next two sections. We will use $S_i = \langle s_{i,1}, s_{i,2}, \ldots \rangle$ to denote this unique decomposition, where S_i is the sequence of system call phrases for the ith replica and $s_{i,j}$ is the jth system call phrase in the sequence, unless otherwise stated. Section 4.4.2 discusses our treatment when there is more than one way of breaking a system call sequence into system call phrases.

We also group repeating phrases in a sequence and consider only one occurrence of such phrase. The objective is not to "penalize" requests that require longer processing. For example, `http` requests for large files normally result in long system call sequences with many repeating phrases.

4.2 ED-based Behavioral Distance

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

To evaluate the evolutionary change between DNA sequences, Sellers [Sel74] proposed a distance measure called *evolutionary distance*, by counting the number of nucleotide changes (including substitutions, deletions and insertions) and summing up the corresponding dis-

tances of substitutions, deletions and insertions. The calculation is easy when nucleotides in the two sequences are aligned properly, i.e., corresponding nucleotides are at the same location in the two sequences. However, it becomes complicated when there are deletions and/or insertions, because the nucleotides are misaligned. Therefore, the correct alignment needs to be found by inferring the locations of deletions and insertions. Figure 4.1 shows an example with two nucleotide sequences and a possible alignment scheme [NK00].

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

Figure 4.1: Example of two nucleotide sequences

Our behavioral distance calculation is inspired by the evolutionary distance method proposed by Sellers [Sel74], where the evolutionary distance is calculated as the sum of the costs of substitutions, deletions and insertions. In behavioral distance calculations, we also have the “misalignment” problem. Misalignment between system call phrases are mainly due to the diverse implementations or platforms of the replicas. For example, the same task can be performed by different numbers of system call phrases on different replicas. Figure 4.2 shows an example with two sequences of system call phrases observed when two replicas are processing the same request. Due to implementation differences, S_2 has an extra system call phrase $\boxed{\text{brk}_2}$ (this phrase has only one system call) which does not perform any critical operation.

$$S_1 = \langle \boxed{\text{open}_1, \text{read}_1}, \boxed{\text{write}_1, \text{close}_1} \rangle$$
$$S_2 = \langle \boxed{\text{open}_2, \text{read}_2}, \boxed{\text{brk}_2}, \boxed{\text{write}_2, \text{close}_2} \rangle$$

Figure 4.2: Example of system call sequences observed on two replicas

To calculate the behavioral distance, we thus need to perform an *alignment* procedure

by inserting dummy phrases so that system call phrases that perform similar tasks will be at the same position in the two aligned sequences. Given a "proper" alignment, we can then calculate the sum of the distances between the phrases at the same position (Section 4.2.1 discusses how we obtain the distances between any two phrases) in the two sequences and use this sum as the behavioral distance.

Given a pair of misaligned system call sequences, there are obviously more than one way of inserting dummy phrases into the sequences. Different ways of inserting them will result in different alignments and hence different behavioral distances between the two sequences. What we are most interested in here is to find the behavioral distance between two sequences when the phrases are aligned "properly", i.e., when phrases that perform similar tasks are aligned to each other. Although it is not clear how to find such an alignment for any given pair of sequences, we know that the "best" alignment should result in the smallest behavioral distance between the two sequences, among all other ways of inserting dummy phrases, because phrases that perform similar tasks have a low behavioral distance, as explained in Section 4.2.1. Therefore, we consider different alignments and choose the one that results in the smallest as the behavioral distance between the two sequences.

Assume that a sequence of system calls S is given in the form of a sequence of system call phrases. Let $\mathsf{prs}(S)$ denote the number of system call phrases in the sequence. Given two sequences S_1 and S_2, we define $\mathsf{Ext}(S_i, n)$ as the set of sequences obtained by inserting $n - \mathsf{prs}(S_i)$ dummy phrases into S_i, at any locations ($i \in \{1, 2\}$). $n = f_1(\mathsf{prs}(S_1), \mathsf{prs}(S_2))$ is the length of the extended sequences after inserting dummy phrases. In order to give more flexibility in the phrase alignments, $f_1()$ ensures that $n > \max(\mathsf{prs}(S_1), \mathsf{prs}(S_2))$. (The definition of $f_1()$ used in our experiments is shown in Section 4.2.3.)

We define the *behavioral distance* between two system call sequences S_1 and S_2 as

$$\mathsf{Dist}(S_1, S_2) = \min_{S'_1, S'_2} \sum_{i=1}^{n} \mathsf{dist}(s'_{1,i}, s'_{2,i})$$

where

$$S'_1 \in \mathsf{Ext}(S_1, n)$$
$$S'_2 \in \mathsf{Ext}(S_2, n)$$
$$s'_{1,i} \text{ is the } i^{th} \text{ phrase in } S'_1$$
$$s'_{2,i} \text{ is the } i^{th} \text{ phrase in } S'_2.$$

The minimum is taken over all possible values of S'_1 and S'_2. $\mathsf{dist}()$ is the entry in the distance table, which defines the distance between any two phrases from the two replicas. (Section 4.2.1 discusses how we obtain the distance table. Here we assume that the distance table is given.)

For example, the calculation of $\mathsf{Dist}(S_1, S_2)$ from the example in Figure 4.2 may indicate that the minimum is obtained when

$$S_1 = \langle \boxed{\text{open}_1,\ \text{read}_1},\ \sigma\ ,\ \boxed{\text{write}_1,\ \text{close}_1} \rangle$$
$$S_2 = \langle \boxed{\text{open}_2,\ \text{read}_2},\ \boxed{\text{brk}_2},\ \boxed{\text{write}_2,\ \text{close}_2} \rangle$$

4.2.1 Learning the Distance table

The calculation of behavioral distance shown above assumes that the distances between any two system call phrases are known. In this subsection, we detail how we obtain the distance table by learning. To make the explanations clearer, we assume that the two replicas are running Linux and Microsoft Windows[1] operating systems.

One way to obtain the distance table is to analyze the semantics of each phrase and then manually assign the distances according to the similarity of the semantics. There are several difficulties with this approach. First, this is labor intensive. (Note that the set of

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

system call phrases is likely to be different for different programs.) Second, the information may not be available, e.g., most system calls are not documented in Windows. Third, even if they are well documented, e.g., as in Linux, the distances obtained in this way will be general to the operating system, and may not work well for the program being monitored. For example, two system call phrases that usually perform different tasks on two platforms may be used by a program to do the same thing.

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

4.2.1.1 Initializing the Distance table

The initial distance values in the distance table play an important role in the performance of the system. Improper values might result in converging to a local minimum, or slower convergence. We introduce two approaches to initialize these distances. We use the first approach to initialize entries in the distance table that involve system calls for which we

know the behavior, and use the second approach for the rest. Intuitively, distance between phrases that perform similar tasks should be assigned a small value.

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

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

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

where

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

Intuitively, if system calls in the two phrases have small similarity values with each other, the distance between the two phrases should be low. (The definition of $f_2()$ used in our experiments is shown in Section 4.2.3.)

The main difficulty of this approach is that Windows system calls are not well documented. We have managed to obtain the system call IDs of 94 exported Windows system

calls with their function prototypes [Neb00].[2] We then assign distances to these 94 Windows system calls and the Linux system calls by comparing their semantics. Since we do not know the system call IDs and semantics of the rest of the Windows system calls, we propose a second method to initialize the distance table for phrases that involve the rest of the system calls.

The second approach to initialize the distance between two phrases is to use frequency information. Intuitively, if two system call phrases perform similar tasks on two replicas, they will occur in the system call sequences in the training data with similar frequencies. We obtain the frequency information when the phrases are first identified by a phrase extraction algorithm and a phrase reduction algorithm; see Section 4.1. The phrase extraction algorithm analyzes system call sequences from sample executions, and outputs a set of system call phrases. The phrase reduction algorithm takes this result and outputs a subset of the system call phrases that are necessary to "cover" the training data, in the sense described below.

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

[2] Nebbett [Neb00] lists 95 exported Windows system calls, but we only managed to find 94, which are not exactly the same as those listed by Nebbett.

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

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

$f_3()$ compares the frequencies with which phrases p_i^L and p_j^W are identified and assigns a distance accordingly. (The definition of $f_3()$ that we use in our experiments is shown in Section 4.2.3.) Distances between a system call phrase and the dummy phrase σ are assigned a constant. $\mathsf{dist}(\sigma, \sigma)$ is always zero.

4.2.1.2 Iteratively updating the Distance table

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

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

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

update $\mathsf{dist}(p_i^L, p_j^W)$ as

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

Intuitively, the larger $\mathsf{occ}_z(p_i^L, p_j^W)$ is, the smaller $\mathsf{dist}_{z+1}(p_i^L, p_j^W)$ should be. (The definition of $f_4()$ used in our experiments is shown in Section 4.2.3.) $\mathsf{dist}(p_i^L, \sigma)$ and $\mathsf{dist}(\sigma, p_j^W)$ are updated in the same way, and $\mathsf{dist}(\sigma, \sigma) = 0$.

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

4.2.2 Real-time monitoring

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

4.2.3 Parameter settings

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

These functions are defined as follows in our experiments:

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

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

4.3 HMM-based Behavioral Distance

One limitation of the ED approach is that it does not take adequate account of the order of system call phrases in each sequence, because the behavioral distance is defined as the sum of distances between aligned phrases. Since system call order is known to be important to detecting intrusions (e.g., [FHSL96, SBDB01, GRS04b, GRS04a]), this is a significant limitation.

Our use of an HMM for calculating the behavioral distance of sequences addresses this limitation. We use a single HMM to model both processes, and so a pair of system call phrases $[s_{1,\cdot}, s_{2,\cdot}]$, 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 call phrases in an observable symbol perform similar tasks, then the probability should be high, otherwise the probability should be low. This probability serves the same purpose as the Dist table in the ED approach. However, in HMM-based behavioral distance, the probability of emitting the same observable symbol is generally different for different states, whereas in ED-based behavioral distance, a universal Dist table is used for every phrase pair in the system call sequences. In this way, our HMM model better accounts

for the order of system calls.

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

In this section, we introduce our Hidden Markov Model and describe how it is used for behavioral distance calculation. We begin in Section 4.3.1 with an overview of the HMM. We then present our algorithm for calculating the behavioral distance in Section 4.3.2, and describe the original construction of the HMM in Section 4.3.3.

4.3.1 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 $V = \{[x, y] : x \in P_L \cup \{\sigma\}, y \in P_W \cup \{\sigma\}\}$ of output symbols. Recall that P_L and P_W are the sets of system call phrases observed on platforms 1 and 2, respectively, and that σ denotes a designated dummy symbol.

- A set $A = \{a_i\}_{0 \leq i \leq N}$ of state transition probability distributions. Each $a_i : \{1, \ldots, N+1\} \rightarrow [0, 1]$ satisfies $\sum_j a_i(j) = 1$. $a_i(j)$ is the probability that the HMM, when in state q_i, will next enter q_j. We will typically denote $a_i(j)$ with $a_{i,j}$. We stipulate that

$a_{0,N+1} = 0$, i.e., the HMM does not transition directly from the start state to the end state. Note that a_i is undefined for $i = N + 1$, i.e., there are no transitions from the end state. Similarly, $a_{i,0}$ is undefined for all i, since there are no transitions to the start state.

- A set $B = \{b_i\}_{1 \le i \le N}$ of symbol emission probability distributions. Each $b_i : (P_L \cup \{\sigma\}) \times (P_W \cup \{\sigma\}) \rightarrow [0, 1]$ satisfies $\sum_{[x,y]} b_i([x, y]) = 1$. $b_i([x, y])$ is the probability of the HMM emitting $[x, y]$ when in state q_i. We require that for all i, $b_i([\sigma, \sigma]) = 0$. Note that neither b_0 nor b_{N+1} is defined, i.e., the start and end states do not emit symbols.

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

1. Initialize λ with q_0 as the current state.

2. Repeat the following until q_{N+1} is the current state:

   (a) If q_i is the current state, then select a new state q_j according to the probability distribution a_i and assign q_j to be the new current state.

   (b) After transitioning to the new state q_j, if $q_j \neq q_{N+1}$ then select an output symbol $[x, y]$ according to the probability distribution b_j and emit it.

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

For the HMM λ we will build, there are many executions that generate the given pair of sequences $[S_1, S_2]$ as in (4.1). We use $\mathsf{Ex}_\lambda([S_1, S_2])$ to denote the set of executions of λ that generate $[S_1, S_2]$. The probability that λ generates the sequences $[S_1, S_2]$, 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 of those sequences. Note that

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

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

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

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

Note that when such an HMM is used to model the system-call behavior of our replicated system, each element of the HMM could be considered as describing a component of the replicated system. For example, a state in the set Q could be considered as describing a task the replicated system needs to perform when serving a client request, a symbol emitted by a state could be considered as describing how the task prescribed by the state is implemented

using system calls. However, the semantics of these elements, e.g., what kind of task the state describes, is not captured by the HMM. These semantics are indirectly represented as the state transition probabilities and symbol emission probabilities.

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

$\Pr_\lambda([S_1, S_2])$ is the probability that system call phrase sequences S_1 and S_2 are generated (in the sense of Section 4.3.1) by the HMM λ, which is used to calculate the behavioral distance between S_1 and S_2. If $\Pr_\lambda([S_1, S_2])$ is greater than a threshold value, the system call sequences will be considered as normal, otherwise an alarm is raised indicating that an anomaly is detected. In this section we describe an algorithm for computing $\Pr_\lambda([S_1, S_2])$ efficiently, given λ, S_1, and S_2. Again, S_1 and S_2 would typically be observed from monitoring the processes. How we build λ itself is the topic of Section 4.3.3.

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

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

To express this algorithm precisely, we introduce the following random variables in an execution of the HMM λ. Random variable State^t is the state after t iterations. (It is undefined if the execution terminates in less than t iterations.) Random variable $\mathsf{Out}_1^{\leq t}$ is the

sequence of system call phrases from P_L in the first components of the emitted symbols (less σ) through t iterations. That is, if in the (up to) t iterations, λ emits $[s'_{1,1}, s'_{2,1}], \ldots, [s'_{1,\ell}, s'_{2,\ell}]$ where $\ell \leq t$, then $\mathsf{Out}_1^{\leq t}$ is the sequence of non-σ values in $\langle s'_{1,1}, \ldots, s'_{1,\ell} \rangle$ (with their order preserved). Similarly, the random variable $\mathsf{Out}_2^{\leq t}$ would be the non-σ values in $\langle s'_{2,1}, \ldots, s'_{2,\ell} \rangle$. Now define

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

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

Base cases:

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

Induction:

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

For example, $\alpha(1,0,i)$ is the probability that q_i is the current state and all that has been emitted is one system call phrase 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 system call phrases 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 call phrases from S_1 and S_2, respectively, have been output, and λ is in some state q_j. (This event occurs with probability $\alpha(u,v-1,j)$.) λ then transitions from q_j to q_i (with probability $a_{j,i}$) and emits $[\sigma,s_{2,v}]$ (with probability $b_i([\sigma,s_{2,v}])$).

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

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

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

Here, $\mathrm{Post}(S, u)$ denotes the suffix of S that remains after removing the first u elements of S. Analogous to the preceding discussion, random variable $\mathsf{Out}_1^{>t}$ is the sequence of system calls from P_L in the first components of the emitted symbols (less σ) in iterations $t + 1$ onward (if any), and similarly for $\mathsf{Out}_2^{>t}$. So, $\beta(u, v, i)$ is the probability of the event that q_i is the current state after some iterations and subsequently exactly the last $l_1 - u$ system call phrases of S_1 are emitted, and exactly the last $l_2 - v$ system call phrases 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 call phrases and HMM states grow.

4.3.3 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 first demonstrate this method for updating the a_i parameters of λ, and then present a similar treatment for the b_i parameters.

4.3.3.1 Refining a_i

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

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

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

That is, $\gamma(u, v, i)$ is the probability of λ being in state q_i after emitting u system call phrases for process 1 and v system call phrases 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 call phrases for process 1 and v system call phrases for process 2, and then transitioning to state q_j, given the entire system call sequences for the processes. Each of these conditional probabilities pertains to one particular subset

of executions that generate S_1 and S_2. As explained in Section 4.3.2, 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 call phrases for process 1 and v system call phrases 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:

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

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 call phrases of S_1 and the first v system call phrases 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,

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

where l_1 and l_2 are the lengths of S_1 and S_2, respectively. Similarly, if $X_{i,j}$ is the number

of transitions from q_i to q_j when generating $[S_1, S_2]$, then

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

With these expectations calculated, we can update the a_i parameters of the HMM λ, using the Baum-Welch method [BP66], as follows:

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

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

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

where w_k is the weight for each pair of sequences $[S_1^{(k)}, S_2^{(k)}]$ in the training set for the current instance of λ. There are many ways of setting w_k [DLC02]. In our experience, different settings affect the speed of convergence, but the final result of the HMM is almost the same. In our experiments, we choose

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

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

The equations above show how the parameters of an HMM can be adjusted in one

refinement. We need many such refinements in order to find a good HMM that generates the training examples with high probabilities. Although more refinements can improve the probabilities, they may also result in overfitting. To detect when to stop the refinement process so as not to overfit the training samples, we use a separate validation set, which also contains pairs of system call sequences recorded from the two processes when processing the same inputs. Briefly, we detect overfitting when the refinement process either decreases $\Pr_\lambda([S_1, S_2])$ for pairs $[S_1, S_2]$ in the validation set or increases the false-alarm rate on the validation set using the alarm threshold needed to detect mimicry attacks (explained in Section 4.4).

4.3.3.2 Refining b_i

The idea of updating b_i parameters of λ is the same as of updating a_i (see Section 4.3.3). Here, we need to calculate the expected number of times λ emits observable symbol $[x, y]$ at q_i, when generating $[S_1, S_2]$. To compute this expectation, we first define a conditional probability, $\zeta([x, y], u, v, i)$, as follows:

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

where

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

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

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

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

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

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

Let the random variable $X_{i,[x,y]}$ be the number of times that state q_i is visited when q_i emits observable symbol $[x,y]$, when λ generates $[S_1,S_2]$. For the same reason as explained in Section 4.3.3,

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

and the b_i parameters of λ can be updated as

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

4.3.4 Implementation issues

There are several implementation issues that deserve comment. First, 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 (in phrases) 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 4.4 using three different web servers on two different operating systems, this guideline yielded values of N between 10 and 33.

Second, 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 third issue concerns the use of a finite set of training samples for estimating the HMM parameters. If we look at the formulas for building the HMM in Section 4.3.3, 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.

4.4 Detection accuracy of ED-based and HMM-based Behavioral Distance

As discussed in Section 4.3, 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 ED-based approach in Section 4.2. In this section, we evaluate an implementation of our anomaly detector using HMM-based behavioral distance to determine whether this is, in fact, true.

Our evaluation system includes two computers running web servers to process client HTTP requests. One of these computers, denoted **L**, runs Linux kernel 2.6.8, and the other, denoted **W**, runs Windows XP Pro SP2. The web server run by each computer differs from test to test, and will be discussed below. In our tests, each of **L** and **W** was given the same sequence of requests (generated from the static test suite of WebBench 5.0,[3] and each recorded the system call sequence, denoted by $S_{\mathbf{L}}$ and $S_{\mathbf{W}}$,[4] respectively, of (the thread in) the web server process that handled the request. The behavioral distance is calculated as described in Section 4.2 and Section 4.3.3.

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

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

[4] We obtain the Windows system call information by overwriting the KiSystemService table in the Windows kernel using a kernel driver we developed.

a good job of discriminating "normal" behavior from even the "best-disguised" abnormal behavior. To compare the results of the ED-based and HMM-based behavioral distance, we presume the same system call sequence an attacker tries to execute, which is simply an `open` followed by a `write`. We use this sequence because it is seemingly the least an attacker must do to modify or create data on the server machine.

Finding the best mimicry attack for the ED approach is relatively easy, because the calculation of behavioral distance is fast and therefore an exhaustive search can be performed. However, we know of no efficient algorithm for finding the best mimicry for the HMM approach (an obstacle an attacker would also face). Therefore, we first propose an efficient algorithm to estimate this best mimicry attack.

4.4.1 Estimating the best mimicry

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

For a fixed pair of system call sequences S_1 and $\hat{S_2}$, let $\hat{\mathrm{Pr}}_\lambda([S_1, \hat{S_2}])$ denote the probability of the most probable execution of λ that generates $[S_1, \hat{S_2}]$. Note that $\hat{\mathrm{Pr}}_\lambda([S_1, \hat{S_2}]) <$

$\Pr_\lambda([S_1, \hat{S_2}])$, since multiple executions can yield $[S_1, \hat{S_2}]$ (including that which occurs with probability $\hat{\Pr}_\lambda([S_1, \hat{S_2}])$). Given S_2, there are many different possibilities for $\hat{S_2}$. Each $\hat{S_2}$ has a corresponding $\hat{\Pr}_\lambda([S_1, \hat{S_2}])$. Here we define the "best" mimicry attack, given S_1, S_2 and λ, as the $\hat{S_2}$ that maximizes $\hat{\Pr}_\lambda([S_1, \hat{S_2}])$, i.e., the estimated-best mimicry attack is

$$\arg\max_{\hat{S_2}} \hat{\Pr}_\lambda([S_1, \hat{S_2}])$$

To summarize, in order to find the estimated-best mimicry attack, we need to try different possible $\hat{S_2}$ sequences, and different executions of the HMM in generating $[S_1, \hat{S_2}]$ in order to find the one that results in the highest probability. Here we propose an efficient algorithm to do this.

We first try to find the estimated-best $\hat{S_2}$, by considering ways to improve a given mimicry attack, i.e., to modify $\hat{S_2}$ to increase $\hat{\Pr}_\lambda([S_1, \hat{S_2}])$. This can be achieved by changing the way a transition is made from any state q_i to q_j when generating $[S_1, \hat{S_2}]$. Since we are modifying an existing mimicry attack, we want to make sure that the modification does not emit any system calls in S_1, otherwise the mimicry attack will fail (though the modification can emit additional system calls for process 2).

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

to find the best way (the one with highest probability), we define

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

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

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

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

where

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

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

With $\{\hat{a}_{i,j}(e)\}$ calculated, the algorithm of finding the estimated-best mimicry attack becomes very similar to the algorithm of finding $\Pr_\lambda([S_1, S_2])$ (see Section 4.3.2). The differences are

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

We define $\delta(u, v, i)$ to be the probability of the most probable mimicry execution to generate exactly the first u system calls of S_1, and exactly the first v system calls of S_2, when the current state is q_i, among all executions. As a technical matter, when computing $\delta(u, v, i)$ inductively, we need to take care to ensure that the HMM executions considered in the calculation of $\delta(u, v, i)$ do not include those that should be considered only in calculating $\delta(u, v', i)$ for $v' > v$. Intuitively, the danger is HMM executions that, in the course of emitting arbitrary system calls before reaching the next attack system call in S_2, in fact insert attack system calls from S_2 as these "arbitrary" system calls. It is for this reason that in calculating $\delta(u, v, i)$ inductively, we need to exclude HMM executions that output elements of S_2 prematurely, hence the arguments to $\hat{a}_{i,j}$ and $\hat{b}_i$. Given this, $\delta(u, v, i)$ can be solved inductively as follows.

Base cases:

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

Induction:

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

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

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

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

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

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

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

An interesting question is whether this algorithm can be extended to find the "real" best mimicry attack. To do so, the corresponding $\delta'(u,v,i)$ needs to be defined as the "highest sum of probabilities of all executions" for (u,v,i). However, in assembling the most probable mimicry as discussed above, do we record $\delta'(u,v,i)$ for one particular $\hat{S}_2$, or $\delta'(u,v,i)$ for all possible $\hat{S}_2$'s? Unfortunately, the latter is required, because when calculating $\delta'()$ of

larger indices, we need the results of $\delta'()$ of lower indices for different $\hat{S}_2$'s. Since for each (u, v, i) we need to record $\delta'(u, v, i)$ for all possible $\hat{S}_2$'s, this algorithm requires exponential computation time and memory in the worst case in the length of the best mimicry. As such, we presently settle for the "estimated-best" mimicry attack, which showed how to compute efficiently above, and leave finding the absolute best mimicry attack to future work.

4.4.2 False-alarm rate when detection the "best" mimicry

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

Once the behavioral distance model is constructed, we find the estimated-best mimicry attack and set the behavioral distance alarm threshold to be the behavioral distance resulting from this estimated-best mimicry, and measure the false-alarm rate of the system that results. A false alarm corresponds to a legitimate request that induces a pair of system call sequences with a probability of emission from λ at most the threshold. The false-alarm rate is then calculated as the number of false alarms divided by the total number of requests. We perform our experiments in nine different settings, defined by the web servers that **L** and **W** are running. (The web servers are Apache 2.0.54, Abyss X1 2.0.6 and MyServer 0.8.)

Table 4.1 presents results using a testing mechanism in which the training (to train the model), validation (to detect overfitting) and evaluation (to evaluate) sets are distinct. System call sequences in these three sets are obtained by sending `http` requests to the web servers using WebBench 5.0[5] and observing system calls made by the web servers when processing these requests.

As stated in Section 4.1, each of these system call sequences usually corresponds to a unique sequence of system call phrases. If there is more than one way to decompose a system call sequence in the training set into system call phrases, we choose to consider only the phrase sequence with the smallest number of phrases. The reason we use the decomposition with the fewest system call phrases is to favor long phrases that correspond to large basic blocks in the program source. If there is more than one way to decompose a system call sequence in the validation set and the evaluation set, or if we are doing online monitoring, we consider all possible decompositions in calculating the behavioral distance, and take the the one yielding the smallest behavioral distance as the result. This is to avoid misclassifying a sequence as an anomaly just because one of the decompositions has a large behavioral distance.

Results in Table 4.1 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.

4.5 Design and implementation of intrusion-tolerant web and game servers

In this section, we present the design, implementation and evaluation of a novel architecture to detect mimicry attacks using behavioral distance. Whereas earlier sections focus on algorithms for computing behavioral distance, here we address the systems issues necessary

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

Server on **L**	Server on **W**	ED-based		HMM-based	
		Mimicry on **L**	Mimicry on **W**	Mimicry on **L**	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 %

Table 4.1: False-alarm rate when detecting the estimated-best mimicry attack

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.

4.5.1 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.

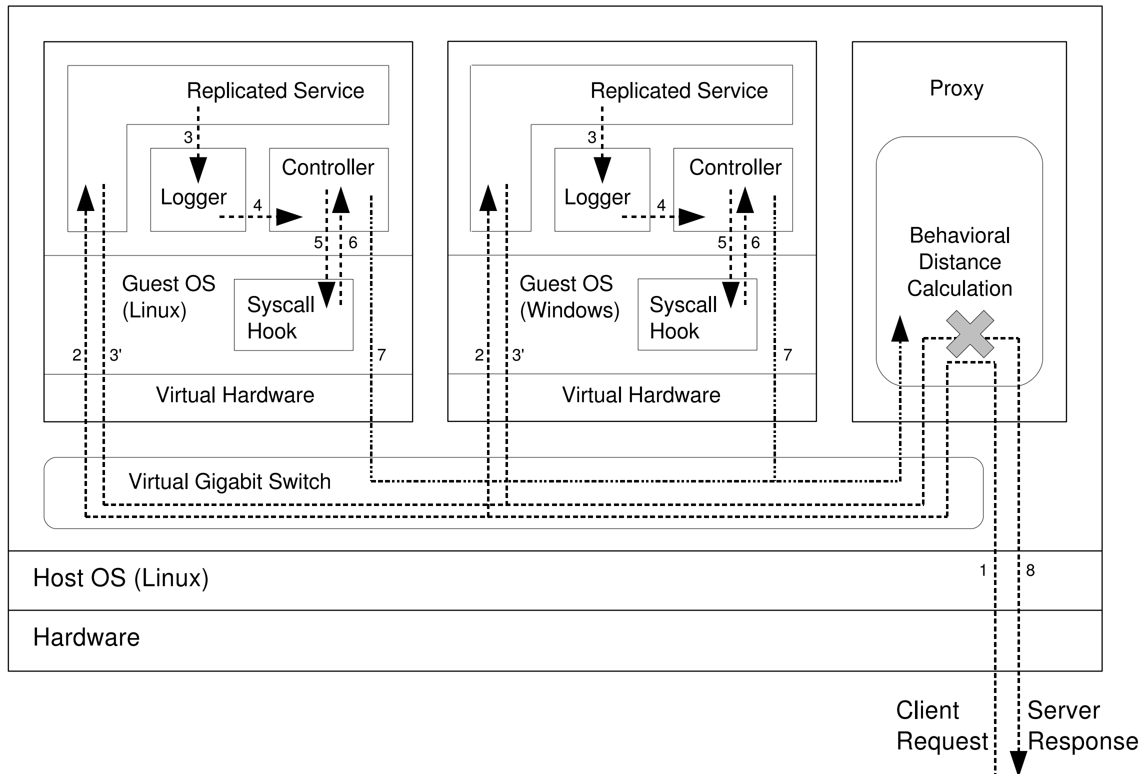
Below we outline the system structure, and then explain the details with two concrete examples—behavioral distance for a web server and an online game server.

4.5.1.1 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.

Figure 4.3 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)



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:	system call info from the kernel
Message 7:	syscall info from the controller
Message 8:	response from the proxy

Figure 4.3: Architecture of the system

to both replicas after some necessary modifications (these modifications are discussed in Section 4.5.1.2). 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.

4.5.1.2 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.

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[6] is used for a user program running as `root` (the controller, see

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

Section 4.5.1.2) 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 call[7] 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 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.

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,

[7]System calls on Windows are also called native API calls or system services.

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.

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 4.4 shows the content of each message exchanged among various components for a client request $\mathsf{req_i}$. Communications among the logger, the controller and the proxy are via UNIX pipes or sockets.

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

$\mathsf{req_i}$	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 system call sequence for process/thread ID k in kernel
$\mathsf{Sys_i}$	The system call sequence for $\mathsf{req_i}$ ($\mathsf{Sys_i}$ is a subsequence of $\mathsf{S_{pid_i}}$)

Figure 4.4: Content of each internal message when processing a client request $\mathsf{req_i}$

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 call[8]. 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.

4.5.1.3 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 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

[8]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 4.5.1.2), with a different parameter. On Windows, we use a new system call that has not been implemented.

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 [ALB06] 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 4.5.2.3) provide a realistic view of the suitability of our approach to similar services written in C/C++, for example.

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.

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.

Implementation issues 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 connections[9]). 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 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`

[9]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.

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 uses 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 4.5.1.3 (32 lines), we have modified less than 1% of the Peekaboom source.

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

4.5.2 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 [TMK02, WS02, KKM+05, GJM06]. We will consider the same type of mimicry attack as discussed in previous sections, in which the attacker tries to make a system call `open` followed by a system call `write`. We also evaluate the performance overhead of the systems when detecting these attacks.

4.5.2.1 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.

4.5.2.2 Web Server

We want to see how the system behaves when serving real web traffic instead of traffic simulated by a bench marking tool as in previous projects [GRS05, GRS06]. 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 [GRS06]. 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 4.5.2.2) 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.[10]

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 [TMK02, WS02, KKM$^+$05, GJM06] 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 [GRS06]) 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.

[10] Apache on Linux and Apache on Windows are different code bases.

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 4.5 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.

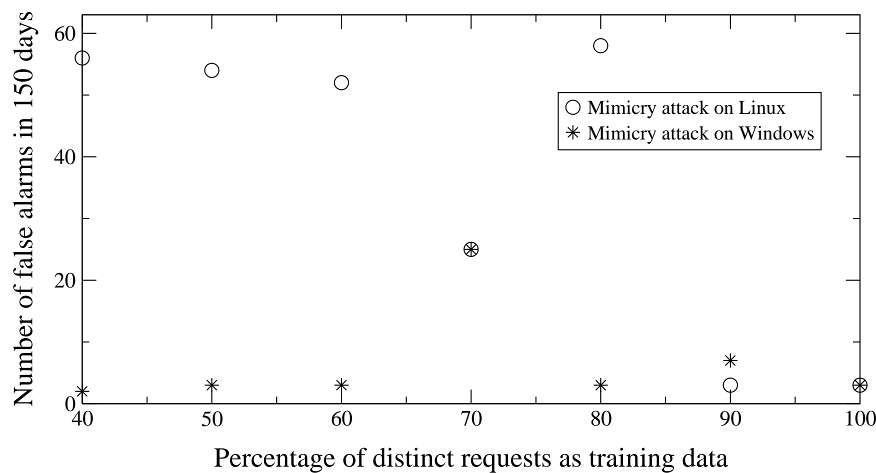


Figure 4.5: Number of false alarms when detecting the “best” mimicry attack

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 are also about an order of magnitude better than those previously reported [GRS05, GRS06]. 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.

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 4.6 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.

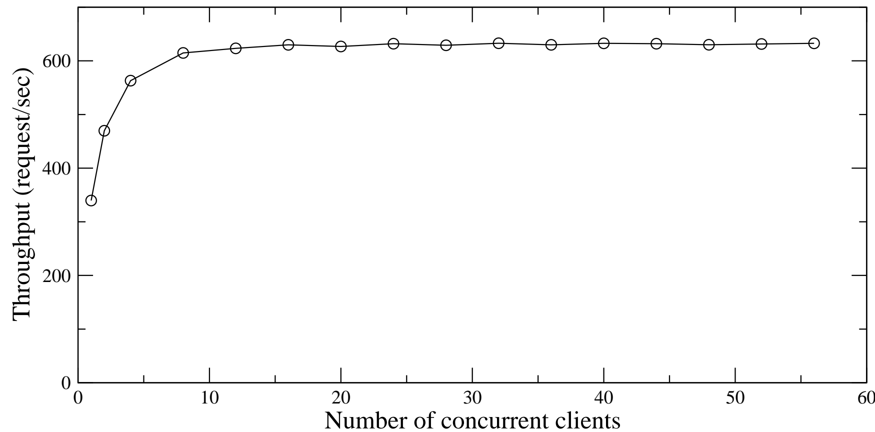


Figure 4.6: Throughput of the web server with different numbers of concurrent clients

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 4.7 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 4.2 shows the average latency measured by the clients on the same local area network.

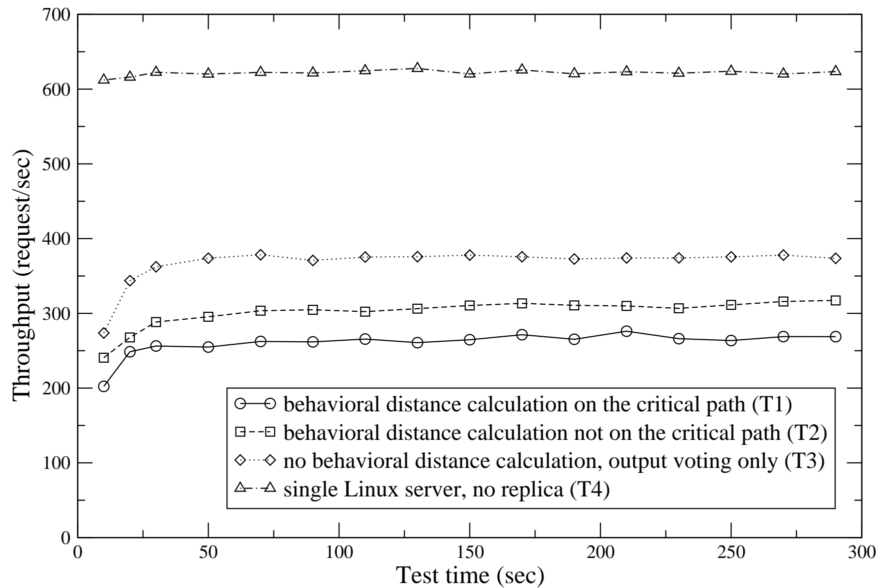


Figure 4.7: Throughput of the web server

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

Table 4.2: Average latency measured by clients

Figure 4.7 and Table 4.2 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 4.8, where L and W denote the replicas running Linux and Windows, respectively.

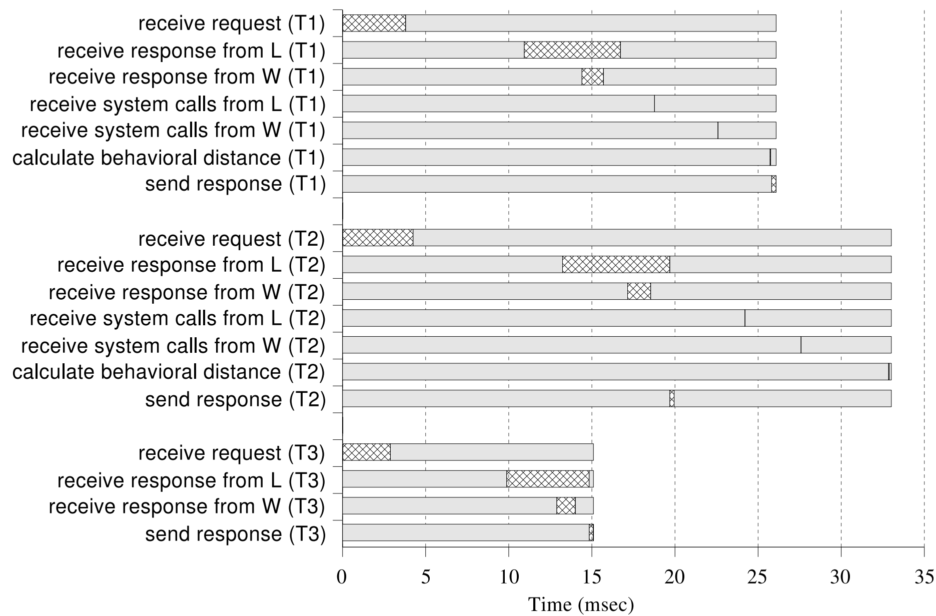


Figure 4.8: Average latency measured by proxy

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) 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 4.8 also confirms earlier predictions [GRS06] 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.

4.5.2.3 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 4.5.1.3). 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.

The game servers on both replicas are compiled using the javac 1.5.0 compiler from Sun. Both replicas run the Java HotSpot server VM 1.5.0.

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 4.5.1.3 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 [GRS06]; 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.

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 4.5.1.3. 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 4.9.

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

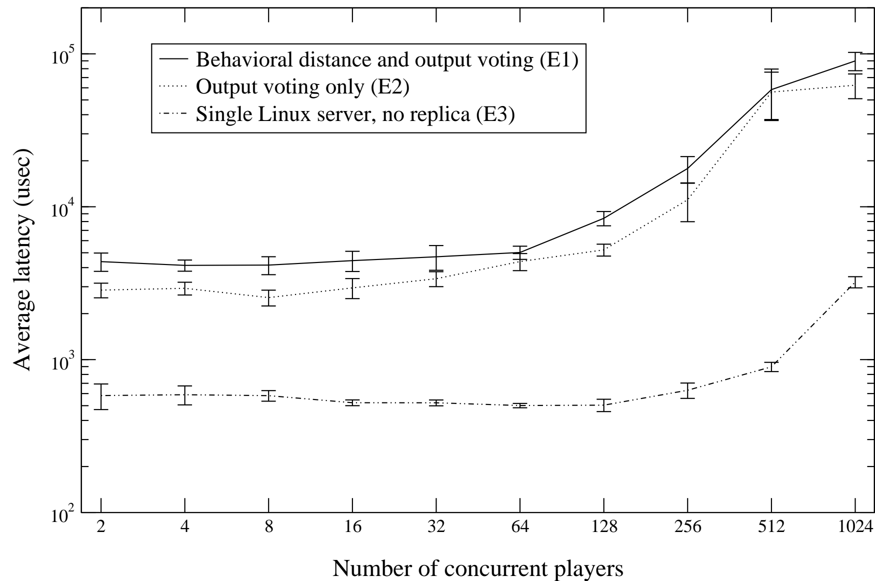


Figure 4.9: Average latency measured by clients on the same LAN

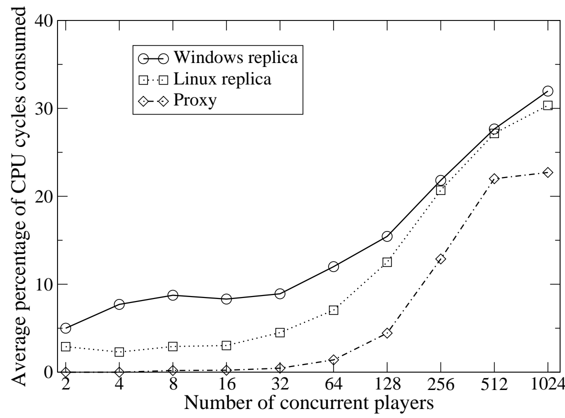
the server is very busy, e.g., when there are 1024 concurrent players, the players experience an additional 86-millisecond latency, which is still hardly noticeable. Also note that the results presented in Figure 4.9 are latencies measured by an automatic player program running on the same local area 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 msecs,[11] 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 4.10 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

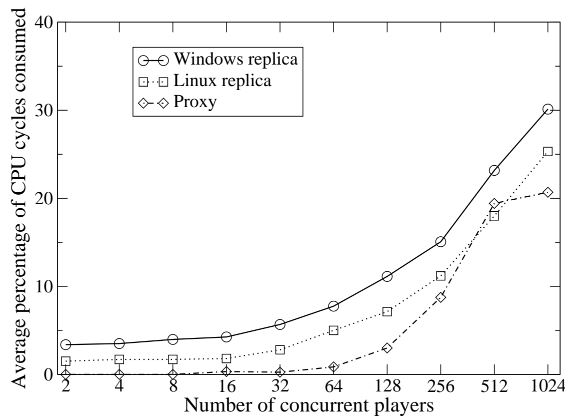
[11] We measured the RTT between a server on our department network (`www.ece.cmu.edu`) and `www.google.com`. Results were between 108 msecs and 119 msecs in 20 runs.

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.

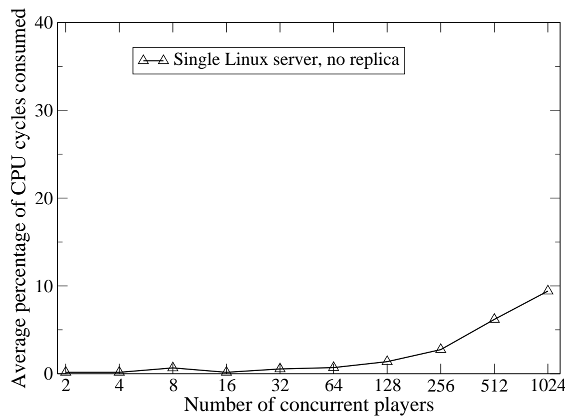
We also monitor the memory usage of the replicas and the proxy. Results show that memory usage is always low in all tests — in most cases less than 10%.



(a) Behavioral distance calculation and output voting (E1)



(b) Output voting only (E2)



(c) Single Linux server, no replica (E3)

Figure 4.10: Average CPU load of the replicas and the proxy

Chapter 5

Conclusion

In this thesis, we explore two novel approaches for more accurate anomaly detection. Execution graph is the first model that both requires no static analysis of the program source or binary, and conforms to the control flow graph of the program. We formalize and prove two properties of the execution graph: (i) it accepts only system call sequences that are consistent with the control flow graph of the program; (ii) it is maximal given a set of training data, meaning that any extensions to the execution graph could permit some intrusions to go undetected. We also evaluate the performance of an anomaly detector using execution graphs.

We propose Behavioral distance to compare the behavior of a process to the behavior of another process that is executing on the same input but that either runs on a different operating system or runs a different program that has similar functionality. Assuming their diversity renders these processes vulnerable only to different attacks, a successful attack on one of them should induce a detectable increase in the "distance" between the behavior of the two processes. We detail two black-box approaches for calculating the behavioral distance and construction the behavioral distance model: one inspired by evolutionary distance (ED) and the second using a new type of Hidden Markov Model (HMM). With

an empirical analysis using three web servers on two different platforms, we show that the HMM-based behavioral distance is able to detect intrusions with a higher accuracy.

We additionally build and evaluate a system, which uses HMM-based behavioral distance to protect Internet servers. We show the detailed implementation of two types of servers: a web server serving static pages and an online game server generating dynamic responses. Evaluation results show that behavioral distance can be used practically to protect these servers. Such a system makes it more difficult for an intrusion to evade detection while generating very few false alarms.

5.1 Limitation of system-call-based anomaly detection techniques

The techniques proposed in this thesis are not able to detect all intrusions. In general, system-call-based anomaly detection techniques have some limitations in detecting intrusions.

First, intrusions are detected by monitoring the system-call behavior of a program; therefore, attacks that do not result in any change in the program's system-call behavior will not be detected. For example, an intrusion that modifies some data on the stack by overflowing a buffer will not be detected unless the modified data causes the system-call behavior to be different from the program's normal execution. In many cases, race conditions may not be detected either. Theoretically, a perfect mimicry attack which makes itself not distinguishable from the program's normal execution cannot be detected, although our techniques using behavioral distance make mimicry attacks more difficult.

Second, we assume that the execution of the program being protected is stable, in terms of its system-call behavior. This is usually true because the behavior of the program is defined by its source code, which is usually static. However, this assumption also makes

the technique inappropriate for self-modifying programs.

5.2 Future work

We have shown that the language accepted by an execution graph is a subset of the language accepted by the corresponding control flow graph. However, we have not evaluated the relative size of the two sets of languages for common programs. In the future, we would like to evaluate the difference between these two sets of languages. This research is interesting because we will then be able to tell how good the execution graph is in approximating the control flow graph. The answer highly depends on the quality of the training data, which is usually called code coverage in program testing. So, another research direction is to define new ways of measuring the quality of training data when used for building the normal behavior model for anomaly detections.

There are also questions in behavioral distance that have not been answered. For example, currently we are only able to tell if there is an intrusion detected, i.e., if the two system call sequences "match" to each other. However, when detecting such an intrusion, we are not yet able to tell which replica is misbehaving. Another interesting research direction is to propose efficient algorithms for measuring behavioral distance when there are more than two replicas.

The behavioral distance measurements we proposed are based on system call numbers observed on the replicas. In the future, we'd like to utilize more runtime information, e.g., the program counters values and the call stack information. For example, we may be able to achieve better detection accuracy if we can apply the idea of execution graphs in measuring behavioral distance.

Bibliography

[ABEL05] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer & Communication Security*, 2005.

[AEMGG+05] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 59–74, October 2005.

[ALB06] 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.

[AMPR01] 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.

[BB93] R. W. Buskens and R. P. Bianchini, Jr. Distributed on-line diagnosis in the presence of arbitrary faults. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 470–479, June 1993.

[BCS06] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[BP66] L. E. Baum and T. Petrie. Statistical inference for probabilistic functions of finite state Markov chains. *Ann. Math. Statist.*, 37:1554–1563, 1966.

[CA78] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proceedings of the 8th International Symposium on Fault-Tolerant Computing*, pages 3–9, 1978.

[CEF+06] 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.

[CH03] S. Cho and S. Han. Two sophisticated techniques to improve HMM-based intrusion detection systems. In *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID 2003)*, 2003.

[CL02] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4), November 2002.

[CP02] 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.

[CRL03] M. Castro, R. Rodrigues, and B. Liskov. Base: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3):236–269, 2003.

[CTL98] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, January 1998.

[DLC02] R. I. A. Davis, B. C. Lovell, and T. Caelli. Improved estimation of Hidden Markov Model parameters from multiple observation sequences. In *Proceedings of the 16th International Conference on Pattern Recognition (ICPR 2002)*, 2002.

[FGH+04] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, 2004.

[FHSL96] 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.

[FKF+03] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, 2003.

[GJM02] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, 2002.

[GJM04] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of Symposium on Network and Distributed System Security*, 2004.

[GJM06] 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.

[GRS04a] 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.

[GRS04b] 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.

[GRS05] 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.

[GRS06] 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.

[JRC+02] 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.

[KKM+05] 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.

[KMVV03] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS 2003)*, 2003.

[Lam78] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[LLC06] L. C. Lam, W. Li, and T. Chiueh. Accurate and automated system call policy-based intrusion prevention. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN06)*, 2006.

[Lu99] X. Lu. A Linux executable editing library. Master's thesis, Computer and Information Science Department, National Unviersity of Singpaore, 1999.

[MD02] I. M. Meyer and R. Durbin. Comparative ab initio prediction of gene structures using pair HMMs. *Oxford University Press*, 2002.

[Neb00] G. Nebbett. *Windows NT/2000 Native API Reference*. Sams Publishing, 2000.

[NK00] M. Nei and S. Kumar. *Molecular Evolution and Phylogenetics*. Oxford University Press, 2000.

[PAC02] L. Pachter, M. Alexandersson, and S. Cawley. Applications of generalized pair Hidden Markov Models to alignment and gene finding problems. *Computational Biology*, 9(2), 2002.

[PC03a] M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2003.

[PC03b] M. Prasad and T. Chiueh. A binary rewriting defense against stack based buffer overflow attacks. In *Proceedings of the USENIX Annual Technical Conference, General Track*, 2003.

[PFH03] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, 2003.

[Pro03] N. Provos. Improving host security with system call policies. In *Proceeding of the 12th USENIX Security Symposium*, 2003.

[Rab89] L. R. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. In *Proceedings of IEEE*, February 1989.

[Rei94] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.

[RF98] I. Rigoutsos and A. Floratos. Combinatorial pattern discovery in biological sequences. *Bioinformatics*, 14(1):55–67, 1998.

[RJL+02] 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.

[RVL+97] T. Romer, G. Voelker, D. Lee, A. Wolman, W.Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of win32/intel executables using etch. In *Proceeding of the USENIX Windows NT Workshop*, August 1997.

[SBDB01] 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.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SDA02] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Proceeding of the Working Conference on Reverse Engineering*, pages 45–54, 2002.

[Sel74] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26:787–793, 1974.

[SR87] K. Shin and P. Ramanathan. Diagnosis of processors with Byzantine faults in a distributed computing system. In *Proceedings of the 17th International Symposium on Fault-Tolerant Computing*, pages 55–60, 1987.

[TMK02] 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.

[TMM05] 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.

[Wag99] D. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, University of California at Berkeley, 1999.

[WD01] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[WDD00] 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.

[WFP99] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[WLAG93] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating System Principles*, 1993.

[WS02] 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.

[YMV+03] 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.