

Anomaly Detection through System and Program Behavior Modeling

Kui Xu

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

Danfeng (Daphne) Yao, Chair
Dennis Kafura
Wenjing Lou
Barbara Ryder
David Evans

November 14, 2014
Blacksburg, Virginia

Keywords: Security, Anomaly Detection, Drive-by Download Detection, Program Analysis,
Machine Learning
Copyright 2014, Kui Xu

Anomaly Detection through System and Program Behavior Modeling.

Kui Xu

ABSTRACT

Various vulnerabilities in software applications become easy targets for attackers. The trend constantly being observed in the evolution of advanced modern exploits is their growing sophistication in stealthy attacks. Code-reuse attacks such as return-oriented programming allow intruders to execute mal-intended instruction sequences on a victim machine without injecting external code. Successful exploitation leads to hijacked applications or the download of malicious software (drive-by download attack), which usually happens without the notice or permission from users.

In this dissertation, we address the problem of host-based system anomaly detection, specifically by predicting expected behaviors of programs and detecting run-time deviations and anomalies. We first introduce an approach for detecting the drive-by download attack, which is one of the major vectors for malware infection. Our tool enforces the dependencies between user actions and system events, such as file-system access and process execution. It can be used to provide real time protection of a personal computer, as well as for diagnosing and evaluating untrusted websites for forensic purposes. We perform extensive experimental evaluation, including a user study with 21 participants, thousands of legitimate websites (for testing false alarms), 84 malicious websites in the wild, as well as lab reproduced exploits. Our solution demonstrates a usable host-based framework for controlling and enforcing the access of system resources.

Secondly, we present a new anomaly-based detection technique that probabilistically models and learns a program’s control flows for high-precision behavioral reasoning and monitoring. Existing solutions suffer from either incomplete behavioral modeling (for dynamic models) or overestimating the likelihood of call occurrences (for static models). We introduce a new probabilistic anomaly detection method for modeling program behaviors. Its uniqueness is the ability to quantify the static control flow in programs and to integrate the control flow information in probabilistic machine learning algorithms. The advantage of our technique is the significantly improved detection accuracy. We observed 11 up to 28-fold of improvement in detection accuracy compared to the state-of-the-art HMM-based anomaly models. We further integrate context information into our detection model, which achieves both strong flow-sensitivity and context-sensitivity. Our context-sensitive approach gives on average over 10 times of improvement for system call monitoring, and 3 orders of magnitude for library call monitoring, over existing regular HMM methods. Evaluated with a large amount of program traces and real-world exploits, our findings confirm that the probabilistic modeling of program dependences provides a significant source of behavior information for building high-precision models for real-time system monitoring. Abnormal traces (obtained through reproducing exploits and synthesized abnormal traces) can be well distinguished from normal traces by our model.

Dedication

*To my parents, Qin Fan and Shouchang Xu;
To my wife, Juan Wei.*

Acknowledgments

I owe my most sincere appreciation to my advisor, Dr. Danfeng (Daphne) Yao. She introduced me to the field of cyber security research, and guided me with not only her profound knowledge, but also great patience, which meant a lot to me. I really enjoyed my work for the past five years. Dr. Yao has always suggested me to have a role model. Over the years of our collaboration, she herself has shown me the important qualities it takes to succeed. She is the best role model I've had, and will always encourage me to be the best person possible.

I would like to thank our department head, Dr. Barbara Ryder. Despite her busy schedule, she always sets time for our weekly project meetings and discussions. Her knowledge in program analysis has been a big help. Dr. Ryder's pursuit for precision makes sure our work is conducted at a high standard. I have a lot of respect for her. I feel honored to have had the opportunity to work with her.

I want to thank all the other committee members, Dr. Wenjing Lou, Dr. Dennis Kafura and Dr. David Evans. Their valuable feedback helps me re-evaluate and improve my dissertation.

I also want to thank many peer students from the department whom I've worked with and learned from over the years: Dr. Huijun Xiong, Dr. Hussain Almohri, Hao Zhang, Xiaokui Shu, Karim Elish, Fang Liu, Ke Tian, Dr. Heshan Lin, Dr. Shucai Xiao, Jing Zhang, Dr. Guanying Wang, Dr. Min Li, Ji Wang. I'm glad to know them and have their company for the years in Blacksburg.

Contents

1	Introduction	1
1.1	Motivations and Challenges	2
1.1.1	Infection Onset Detection	2
1.1.2	Quantitative Program Behavior Modeling	3
1.2	Contributions	4
1.3	Document Organization	7
2	Background and Related Work	8
2.1	Infection Onset Detection	8
2.1.1	Malware and Drive-by Download Detection	8
2.1.2	System Integrity Solution	9
2.1.3	Browser Security Solution	11
2.2	Quantitative Program Behavior Modeling	11
2.2.1	Learning-Based or Hybrid Flow-Sensitive Models	11
2.2.2	Program Analysis-Based Flow-Sensitive Models	12
2.2.3	Techniques Improving Context Sensitivity	13
2.2.4	Consistency in Control Transfer	13
2.2.5	Data Flow Modeling	15
2.2.6	Mimicry Attack	15
3	Drive-by Download Attack Detection	17
3.1	Models And Definitions	17

3.2	DeWare Design and Its Components	19
3.2.1	File-System Monitor: Confining Applications' Downloads	20
3.2.2	Real-Time Execution Monitoring on Host	21
3.2.3	Input Recorder	22
3.2.4	Dependency Rules	22
3.2.5	Piggybacking Download and its Detection	24
3.3	Prototype Implementation in Windows	24
3.3.1	File-System Monitor and Execution Monitor	24
3.3.2	Collect and Correlate User Behavior Information	26
3.4	Experimental Evaluation	28
3.4.1	Detecting Known DBD Exploits and Real-World Malicious URLs	28
3.4.2	User Study	29
4	Quantitative Modeling of Program Call Sequence Behavior	37
4.1	Overview of Our Approach	37
4.1.1	Hidden Markov Model	38
4.1.2	A Motivating Example	39
4.1.3	Design Overview and Workflow	41
4.1.4	Probability Definitions	42
4.2	Probability Forecast of Call Sequences	44
4.2.1	Reachability Probability	45
4.2.2	Call-Transition Matrix of Function	46
4.2.3	Aggregation of Call Transitions	49
4.2.4	An Aggregation Example	54
4.2.5	HMM Initialization	56
4.3	Evaluation of Statically Initialized Model	57
4.3.1	Experimental Setup	57
4.3.2	Dynamic Model Training and Testing	59
4.3.3	Impact of Segment Length	62

4.3.4	Classification Accuracy	63
4.3.5	Impact of Hidden States on Detection	66
4.3.6	Detection of Code-Reuse Exploits	67
4.3.7	Running Time Information	71
4.3.8	Summary of Experimental Findings	72
5	Quantitative Call Sequence Modeling with Context Sensitivity	75
5.1	Context-Sensitive Model Overview	75
5.2	Probability Forecast of Context-Sensitive Call Sequences	79
5.2.1	Operations of Probability Processing	79
5.2.2	HMM Initialization	82
5.3	Experimental Evaluation	83
5.3.1	Experiment Setup	83
5.3.2	Classification Accuracy	86
5.3.3	Detection of Real-World Exploits	93
5.3.4	ROP Difficulty Under Context-Sensitive Model	95
5.3.5	Summary of Experimental Findings	97
6	Conclusions and Future Work	99
6.1	Conclusions	99
6.2	Future Work	100
	Bibliography	101

List of Figures

3.1	Schematic drawing of the work flow for detecting the infection onset on a host based on analyzing user-behaviors and file system/process properties.	19
3.2	The system framework of DeWare implementation.	25
3.3	JavaScript code for parsing save-file dialog and saving attributes associated with the user's mouse-click action.	32
3.4	An example of semantic correlation.	33
3.5	DeWare's Detection algorithm in pseudo-code	34
3.6	Histogram on the interval in milliseconds of the user click events and their corresponding file creation events.	35
3.7	(a) The averaged false positive rate vs. various threshold values based on a total of 25,092 file-creation events including temporary files by the browser. (b) Reduced false positives with semantic-based comparison between user's mouse-click events and downloaded files.	36
4.1	Illustrations of classification deficiencies in program behavior models that are constructed from static program analysis (left) or program traces (right). Suppose there exist three statically feasible execution paths P_1 , P_2 , and P_3 , among which paths P_1 and P_3 are much more frequent to occur than the rare path P_2 . The paths represent the system call sequences.	40
4.2	A diagram illustrates our anomaly detection workflow.	41
4.3	Examples of conditional probabilities and reachability probabilities for function $f()$. Conditional probability of a node pair is shown on the edge. Reachability probability of a node is shown in the node. ϵ and ϵ' represent the external call site and return site of $f()$	43
4.4	Illustration of call sequences in a caller function $g()$ and a callee function $f()$. Indices are from topological sort. The AGGREGATION operation replaces and expands entries with $f()$ in $g()$'s call-transition matrix with calls in $f()$	51

4.5	CFG for caller function $g()$. Function $f()$ is called within function $g()$	55
4.6	Part of the control flow for a function.	59
4.7	HMM classification accuracy in terms of the probability distribution of normal and abnormal syscall segments of lengths 15 in (a)(c) and 9 in (b)(d) for gzip in our converged models. (c) and (d) give a zoomed-in look of (a) and (b) respectively. X-axis represents probability ranging from 0 to 1. Y-axis represents the number of segments. Normal segments should have high probabilities, whereas abnormal segments should be low probability.	62
4.8	Comparison of STILO and regular HMM's false negative rates (in Y-axis, base-10 log scale) for program bash on system calls and library calls under the same false positive rates (in X-axis).	64
4.9	Comparison of the averaged false negative rates (in Y-axis) across the six programs from SIR on system call or library calls by our model and the regular HMM with respect to the false positive rates (in X-axis). Standard errors are shown.	64
4.10	Comparisons of classification accuracy in our model and the regular HMM for library and system calls. X-axis shows false positive rates (misclassified normal segments). Y-axis (logscale, base 10) shows false negative rates (misclassified abnormal segments).	65
4.11	The (unpredictable) impact of the number N of hidden states on classification accuracy. Classification results of system calls by regular HMM with various numbers of hidden states for grep and gzip programs are shown, as well as results from STILO HMM. For each HMM, N is shown as multiples (X) of the number of distinct calls in traces.	67
4.12	Comparison between our model and regular models with various numbers of hidden states (The legends in the figure indicate the number of states used, we use x to represent the number of states used in most basic regular model which is the number of distinct calls in trace). X-axis is the false positive rate (normal segments miss-classified), and Y-axis is the corresponding false negative rate (abnormal segments misclassified).	68
4.13	Numbers of iterations needed for HMM models to converge for library calls and system calls in our models and regular models. STILO models require much less training iterations than regular models.	72
5.1	Examples of conditional probabilities and reachability probabilities for function $f()$. Conditional probability of a node pair is shown on the edge. Reachability probability of a node is shown in the node. ϵ and ϵ' represent the external call site and return site of $f()$	76

5.2	Example control flow graph for function $g()$, which is a caller function of $f()$.	77
5.3	Example of a normal and abnormal call sequences under a code-reuse attack. Caller information is associated to each call as the context information. . . .	77
5.4	Comparison of STILO and Regular HMM's false negative rates (in Y-axis, base 10 log scale) for server programs proftpd and nginx on library calls under the same false positive rates (in X-axis). Clustering is applied to the initial model of proftpd	87
5.5	Comparison of STILO and Regular HMM's false negative rates (in Y-axis, base 10 log scale) for server programs proftpd and nginx on system calls under the same false positive rates (in X-axis).	88
5.6	Comparison of the averaged false negative rates (in Y-axis) across the eight programs tested on system call or library calls by our basic and context-sensitive STILO models and the regular HMM, with respect to the false positive rates (in X-axis). Standard errors are shown.	88
5.7	Numbers of iterations needed for HMM models to converge for library calls and system calls in our basic and context-sensitive STILO models and regular HMM model. STILO models require less training iterations than regular models.	91
5.8	Comparison of STILO and Regular HMM's false negative rates (in Y-axis, base 10 log scale) for evaluated utility programs on system calls under the same false positive rates (in X-axis).	93
5.9	Comparison of STILO and Regular HMM's false negative rates (in Y-axis, base 10 log scale) for evaluated utility programs on library calls under the same false positive rates (in X-axis). Clustering is applied to the initial models of proftpd , vim and bash	94

List of Tables

2.1	Comparison of selected drive-by-download solutions.	10
2.2	Comparison of different host-based anomaly detection approaches with program behavior modeling.	14
3.1	User inputs recorded through our Firefox extension.	27
3.2	Comparison of off-the-shelf security products' effectiveness against IE 7 XML DBD attack [54].	30
4.1	A call-transition matrix of the function in Figure 4.3 under the uniform distribution for conditional probabilities. ϵ represents the external caller of this function. ϵ' represents the external return site.	47
4.2	Examples of immediate call sets.	49
4.3	Statistics of the size of programs in experiment.	57
4.4	Statistics of programs and test cases used in experiment. The collected traces are broken into 15-grams (segments) for classification.	58
4.5	Description of applications used in experiment.	60
4.6	Numbers of hidden states used in our HMM and the regular HMM for library or system calls. For regular model the number of hidden states used is the number of distinct calls in observed traces, n_r for libcall and n'_r for syscall. For our model it is the number of distinct calls in our aggregated call-transition matrix, n_o for libcall and n'_o for syscall.	66
4.7	STILO successfully detects <i>Abnormal-A</i> segments from real-world exploits. It recognizes attack sequences without triggering any false alerts on <i>Normal</i> segments.	69
4.8	STILO model generates zero or ultra low probabilities on abnormal segments during a subtle instruction-reuse ROP exploit. In comparison, these segments are not recognized by the regular HMM as abnormal (false negatives). . . .	69

4.9	STILO analysis runtime in seconds. CFG is for CFG constructions. Prob. Est. is for probability estimation in functions. Aggr. is for the aggregation of call-transition matrices.	71
5.1	A call-transition matrix of the function in Figure 5.1 under the uniform distribution for conditional probabilities. ϵ represents the external caller of this function. ϵ' represents the external return site. Each column or row represents a system or library call associated with its caller function as context information.	80
5.2	A call-transition matrix of the caller function in Figure 5.2 under the uniform distribution for conditional probabilities. ϵ represents the external caller of this function. ϵ' represents the external return site. Each column or row represents a system or library call associated with its caller function as context information.	81
5.3	The call-transition matrix for caller function $g()$ after aggregation. Each column or row represents a system or library call associated with its caller function as context information.	81
5.4	Statistics of library calls in programs and traces.	84
5.5	Statistics of system calls in programs and traces.	84
5.6	Averaged accuracy improvement of STILO models over regular HMM models. The accuracy of each model is measured by its false negative rate averaged over tested programs and evaluated false positive rates, as in Equation 5.3.	89
5.7	Average number of training iterations needed for an HMM to converge. Comparisons are among our basic and context-sensitive STILO models and regular HMMs.	90
5.8	Statistics about the clustering of calls in the initial HMM models for selected programs and models and the estimated speedup.	91
5.9	Statistics about termination set. The termination set is part of the traces which is kept aside and used during training to prevent model overfit.	93
5.10	STILO successfully detects <i>Abnormal-A</i> segments from real-world exploits.	94
5.11	Search result of useful [SYSCALL...RET] gadgets for making a sequence of system calls. Search includes evaluated binaries and libc.	96
5.12	Analysis runtime for STILO-context model in seconds. CFG is for CFG constructions. Prob. Est. is for probability estimation in functions. Aggr. is for the aggregation of call-transition matrices.	98

Chapter 1

Introduction

Various vulnerabilities exist in the applications we rely on, which are caused by reasons such as human mistakes, software design flaws and many others. These vulnerabilities could become easy targets of attackers. As a result, malware can get downloaded and installed on a personal device stealthily without users notice and permission, this type of drive-by-download (DBD) attacks has become a great threat to online security. Recently, we've also seen more and more successful high-profile attacks aimed at big name companies in various business areas, such as Adobe [2], Target [110], Nasdaq [76]. Sensitive information of customers is breached including name, email, password, phone number, credit card and so on.

During the exploitation on a vulnerable program, control-flow hijacking is usually observed where a victim program executes the attacker's code injected through a vulnerability in the program. Newer generations of code-reuse based hijacking techniques even allow attackers to compose the attack control flows from the victim's program's code in the memory. For example, `return_to_libc` and `return_oriented_programming` (ROP) exploits reuse and reorder the existing code (e.g., library calls, machine instructions) in the victim program's memory to realize attack sequences. An attacker can also compose new system call sequences from the existing set of legitimate calls to perform malicious activities. Because of their subtlety and stealth, these code-reuse exploits pose challenges to the detection and monitoring mechanisms. For example, unlike the conventional stack-smashing exploits, code-reuse exploits do not execute any code from the memory stack; thus the widely adopted non-executable stack mechanism is defenseless.

Conventional signature-based techniques may not be effective against zero-day exploits or code with sophisticated obfuscation, and the increasing stealth in modern exploits demand sophisticated and precise program behavior modeling and runtime classification.

In the remaining of this chapter, we first discuss the problems we are facing and the research challenges. Then we present our technical contributions and approaches to solve

these problems.

1.1 Motivations and Challenges

Any Internet-connected machines or devices (smartphones, PCs, servers) could be the potential targets of an attacker. Vulnerable applications running on both client and server side systems are prone to malware infection, which is a big threat for information security and privacy. It's important to be able to identify the infection onset, which is an effective approach for keeping out the attack and the delivery of malware content. Rules can be defined to filter out abnormal or suspicious events.

As attacks are constantly evolving to become more stealthy and sophisticated, it also makes the identification of intrusions harder. A more general approach is to characterize the normal behaviors of applications. Based on the program execution behaviors, normal model can be learned, which is able to catch any deviations of behaviors from the normal patterns. It covers an attack at various stages beyond the initial infection phase.

1.1.1 Infection Onset Detection

We start with investigating the security of user side applications and the anomaly detection when they are infected. Among all the attacks, malicious software (malware) downloaded from the Internet has been the leading infection vector [24]. Malware may be delivered stealthily through a networked application such as a browser, a peer-to-peer file sharing client, or a chat application. Web browser is the most common vehicle for a host to contract malware. Provos *et al.* analyzed the content of several billion URLs and executed an in-depth analysis of approximately 4.5 million URLs. From the set, about 450,000 URLs were found successfully launching drive-by-downloads of malware binaries and another 700,000 URLs also seemed malicious [80]. Drive-by-download (DBD) attacks exploit software or design vulnerabilities in a browser or its external components, and stealthily fetch executables from remote malware-hosting server without any user permission. Botnets often use DBD as the initial infection vector, e.g., Torpig [106]. Other networked applications besides the browser may be vulnerable to drive-by-download attacks. For example, a proof-of-concept Quicktime-based drive-by-download attack has been demonstrated in Second Life [81]. Unauthorized web download and process creation are among the symptoms of malicious bots studied by Morales *et al.* [72].

Conventional signature-based techniques may not be effective, especially against zero-day exploits or when obfuscation is used. In comparison, host-based detection approaches are much more feasible against drive-by download attacks and the onset of infection in general.

One of the challenges here is how to distinguish drive-by downloads from legitimate down-

loads caused by human users. Another issue is how to efficiently handle the *voluminous* application-triggered benign downloads, which are not directly caused by user actions. We refer to these downloads as *indirectly* caused by the user’s action. For example, when a user types `www.cnn.com` in the address bar, the browser fetches the main page `index.html` as well as any objects embedded in that page, including JavaScript files, images. It automatically creates temporary files *without* explicit user requests.

Our approach is to strategically monitor file-system events and analyze them with observed user actions. We aim to identify the dependency between system events and user activities. Our goal and challenge is how to design an access control component and policies within the operating system that enforce and regulate behaviors of applications in terms of accessing system resources.

1.1.2 Quantitative Program Behavior Modeling

Both client side and server side applications suffer various attacks from the Internet. It is desirable for a general anomaly detection tool to provide quantitative measurements to predict the real-time assurance of a system or application. We define assurance as the evidence-based confirmation that the system or software is behaving as intended and has not been tampered with by malicious entities (e.g., malicious software or malicious insider). The capability to efficiently compute the likelihood of each system event occurring is important.

In order to perform anomaly detection, one builds models to represent the normal execution patterns, which are used to detect anomalous program behaviors due to control-flow hijacking, unexpected inputs, or operational errors. There exist several learning approaches for building the normal program behavior models from execution traces. A straightforward-yet-unscalable method is to collect n -grams of program call traces (e.g., system calls) to compose a set of allowable call sequences. This n -gram approach has been used to analyze system calls [38, 52, 119] and library calls [57]. Any sequence with new call or out-of-the-order calls is classified as an anomaly. The n -gram approach needs to enumerate all possible call sequences, and is not efficient.

There exist several scalable learning techniques for building program behavior models, for example the automaton model [66, 91], the hidden Markov model (HMM) [44, 117, 124], as well as the execution-graph model [42]. These models represent the allowable control flow or call transitions, supporting **flow-sensitive** detection. Flow sensitivity refers to the model’s ability to represent and analyze the order of execution of statements in the program. Because of the underlying support for modeling and computing conditional probabilities, the hidden Markov model is more advantageous than the regular automaton model, capable of providing the maximum likelihood associated with a call sequence occurring. Thus, HMM tells not only whether a call sequence is feasible to occur or not, and also how likely it occurs in the normal program execution.

However, program behavior models constructed solely through learning from program traces (e.g., [91, 117]) skew toward the (limited) training data, hurting the detection accuracy. For modern complex software it is extremely challenging to obtain traces with close-to-full branch, statement, or def-use coverage. It is typical to have 50-60% coverage for a test-case generation tool [94, 111]. Incomplete training data results in excessive false alarms in a learning-based anomaly detection system, as legitimate call sequences not seen in the training set may not be recognized.

Unlike learning-based models, program behavioral models developed through static code analyses on control flows ([49, 112]) are *complete* in that all the statically feasible paths can be predicted. However, because of the lack of run-time information, statically constructed behavioral models cannot distinguish path frequencies. Paths with different occurrence frequencies are indistinguishable. This lack of quantification in static program modeling can cause important signs of run-time program misuses or undesirable program-behavior changes to be ignored. Binary-transformation based control-flow integrity (CFI) techniques are not designed to offer quantitative behavior classification, either.

Among the above mentioned approaches, [38, 52, 119, 44, 117, 124] are all flow-sensitive, as they only capture the order of system or library calls made by a program. Context sensitivity is the ability to recognize different calling context associated with a call, when collecting and monitoring program traces. Adding context information to the behavior model has been shown to improve the precision. Existing context-sensitive models can be built either dynamically [91, 34] or statically [49, 112, 33, 46, 47].

Context information increases the precision of program behavior modeling. However, without run-time information, these models are not able to distinguish path frequencies, thus can not give quantitative measurement. Also, maintaining the representing automata and reducing runtime overhead has always been a challenge.

Therefore, we want to combine and balance all the advantageous features in the design of our program behavior model with the following goals:

- To associate context information to each call in our model for increased model precision.
- To achieve quantitative reasoning on program behavior occurrences.
- To cover both static and dynamic control-flow behaviors.

1.2 Contributions

As discussed in Section 1.1, our focus in this dissertation is host-based program anomaly detection, and we mainly address two major problems:

1. We investigate the security of client side applications, and study the problem of providing an anomaly detection system that detects the application infection onset. We achieve this by defining and enforcing the dependencies between user actions and system events.
2. Advanced exploitations and attacks are getting more stealthy, thus more accurate detection model is needed to capture the normal behaviors of a running application. We take the idea of integrating the information from both static analysis and dynamic training, and provide a quantitative, flow&context-sensitive and more accurate model for program anomaly detection.

In the following we highlight the contributions of this work.

1. *Infection Onset Detection:*

We give new definitions for direct and indirect causal relationships among user actions and system events in the context of host security. Identifying and enforcing causal relationships are useful in detecting the onset of infection. We point out the associated technical challenges and usability requirements. We show that the integration of human behaviors can greatly improve the design and effectiveness of cyber security solutions.

- (a) **We present our design and implementation of a tool called *DeWare* that detects the onset of infection including drive-by downloads.** DeWare identifies the infection onset by detecting download-and-execute patterns that most stealthy malware exhibits. Our work uniquely incorporates user-behavior characteristics in generating and evaluating policies used for detecting attacks. Our behavior-based monitoring approach is general, and is useful beyond the specific drive-by download problem studied.
- (b) **We define rules for identifying dependency between the system events and the user actions that initiate them.** Finding causal relations among multiple sets of events typically requires sophisticated statistical inference techniques [6, 40]. In operating system, however, system events are created in response to user actions. We demonstrate that simple-yet-effective rules can be defined for enforcing the correct system characteristics and securing a host. Our approach can be generalized to other paradigms of security such as enforcing dependencies among user inputs and outbound traffic.
- (c) **We perform experiments** including a user study with 21 participants to evaluate DeWare’s ability to detect DBD exploits both in a lab setting and 84 malicious websites in the wild; the ability to accurately identify legitimate user downloads with low false positives (<1%). DeWare generates zero false alarm when being automatically tested on 2000 legitimate websites.

2. *Quantitative Program Behavior Modeling:*

We present a classification technique that can detect anomalous program executions and call sequences. The classification is based on a probabilistic control-flow model representing the expected call sequences of the program. Our construction of this probabilistic control-flow model is new. It incorporates both statically and dynamically extracted control-flow information.

We design a rigorous probability representation to model the statically extracted control-flow graph and call graph information of a program (e.g., call transition and branching factor). We use these probabilities to customize our classification model (namely the hidden Markov model). Our experiments show that this static customization significantly boosts the quality and coverage of the learner. Our detection system is easy to deploy. Compared to many existing approaches, it does not require any binary transformation of the program. Our contributions are summarized as follows.

- (a) **We present a probabilistic control-flow model and a series of algorithms that transform statically generated control-flow graphs and their corresponding call graph to a rigorous probabilistic representation.**

This probabilistic representation of the program’s call transitions is used to strategically initialize the hidden Markov model (HMM), in order to achieve high run-time anomaly-based detection accuracy. Our method organically integrates probabilistic reasoning with static predictions of feasible paths, resulting in significantly improved system assurance.

- (b) **We further include the caller function of each library/system call as the context information, and the resulting model is both flow-sensitive and context-sensitive.**

Our approach improves the precision of a program behavior model in two dimensions: i) Improved flow-sensitivity enforces the correct order call sequences are executed. ii) Context-sensitivity makes sure system/library calls are made from the legitimate caller context.

- (c) **Our prototype – referred to as *STILO* – is capable of analyzing and classifying both system call and library call traces of C/C++ programs in Linux OS.**

STILO stands for **ST**atically **I**nitialized mark**O**v. We extensively compare the classification accuracy and performance of STILO with regular HMM models. Our evaluation is performed on more than 4,000 test cases from eight Linux applications including utility and server programs.

- (d) **STILO consistently outperforms the regular HMM models in classification accuracy, for both flow-sensitive and context-sensitive models, on both library and system call monitoring.**

STILO detects all the code-reuse exploits evaluated, including subtle ROP and `return_to_libc` attacks involving legitimate calls. The detection is successful

without triggering any false positives in normal program traces.

Our experimental findings suggest that reasons for STILO’s improved accuracy are two-fold: **i)** an informed set of initial HMM probability values (transition and emission probabilities and probability distribution of hidden states) and **ii)** a more optimized number of hidden states. Both items are crucial – STILO outperforms the regular HMMs with the similar number of hidden states. This finding suggests the effectiveness of our program-analysis-guided probability initialization in boosting the detection.

We demonstrate the combination of both static and dynamic program information, and also the integration of context-sensitivity into one program behavior model. This new modeling technique achieves high anomaly detection accuracy, advancing the state-of-the-art in program-behavior-modeling based anomaly detection and providing more effective tools for cyber defenders in battling against modern stealthy exploits.

1.3 Document Organization

In the rest of the dissertation, in Chapter 2, we discuss the necessary background knowledge of our research and the related work. Chapter 3 presents our approach for drive-by download attack detection. We introduce a more general probabilistic anomaly detection method for program behavior modeling in Chapter 4. We further extend our work by adding calling context to our probabilistic model in Chapter 5. Chapter 6 concludes our work and discusses future directions.

Chapter 2

Background and Related Work

This chapter describes the background of our research work, gives an overview of the related literature and how our work fits in. Section 2.1 reviews various malware intrusion detection approaches, and mainly covers solutions for client-side application and system protection. Section 2.2 focuses more on the work that perform anomaly detection by modeling program execution behaviors.

2.1 Infection Onset Detection

Jaeger and colleagues did pioneering work in operating-system security and security kernel architecture for OS-level control of program behaviors, including regulating downloaded executable content [56] and general-purpose policy enforcement through intercepting inter-process communication in OS [55]. In comparison with existing OS-security work, our work is more specific but not limited to the browser environment. The novelty of our solution is the incorporation of user-behavior characteristics and the leveraging of system-event dependency in evaluating security policies.

2.1.1 Malware and Drive-by Download Detection

Cova, Kruegel, and Vigna proposed a DBD detection solution [22] that abstracted and categorized commonly shared features in the Javascripts that launch drive-by-download attacks. Another DBD-detection solution was proposed based on monitoring browser’s inter-module communication patterns [98]. The methods used in [22, 65, 83] are based on feature extraction and classification from malicious code such as existence of redirection and obfuscation. Careful preparation and selection of features make this feature-based detection robust against known threats. [73, 116] both utilize virtual machine to load webpage, take records and per-

form post-analysis. This class of behavior-based detection is better at catching 0-day attack. Instead of implementing a client-side protection, placing detection on a proxy as a network service helps reduce the client side overhead and also increase efficiency [73, 83]. In [31], string buffers are checked for executable instructions, which enables the solution to detect the shellcode before an vulnerability can be exploited. Egele, Kirda, and Kruegel [30] gave general introductions to drive-by exploits and mentioned possible detection and mitigation approaches (without concrete implementations).

The work that is conceptually close to ours is BLADE [68], which is an effective host-based solution independently developed by Lu *et al.* for detecting drive-by downloads, in particular unconsented-content execution [68]. The authors performed extensive experiments in Windows OS environments. Although sharing the same goal, our technical approaches differ significantly, particularly on how to obtain user behavior information and how file system is monitored. DeWare treats applications as a black box – passively monitors the operating system and does not change to how applications access the existing file systems. BLADE performs redirections on browsers’ I/O requests. Our solution is designed for general applications, not specific for the browser; thus we provide monitoring mechanisms in the kernel level, which are external and independent of the application. DeWare functions well even without the optional semantic comparison that involves an application-level component. Our experimental evaluation approach is also different, including a nontrivial user study with 21 participants.

A comparison of related drive-by download detection work is in Table 2.1.

The Alcatraz work [64, 107] also aims at malware defense, but by detecting the kind of malware that infects victim machines through user permitted installation. Botzilla [84], on the other hand, detects malware at a later point after the infection. Their approach is based on the analysis of traffic pattern when a malware initiates contact with its maintainer.

2.1.2 System Integrity Solution

There exist several system integrity solutions that may bear superficial similarity to our work. Tripwire [101] is a cryptographic-based solution that aims to detect tampering in files by comparing the cryptographic hash values of file systems. Baliga, Iftode, and Chen proposed a rootkit containment strategy Paladin [9] that is based on tracking processes’ hierarchical relationships and their corresponding file creation, and using policies to restrict processes’ privileges to directories and memory access. What sets our work apart is that i) we correlation user inputs with file system monitoring; and ii) we focus on distinguishing authorized and illegal file creation. It is worth mentioning that in Paladin virtual machine monitor (VMM) is used to enforce the integrity of the detection, namely policy tables. VMM can be applied in our model to relax the assumption of the trust on the operating system.

Table 2.1: Comparison of selected drive-by-download solutions.

Solutions	Architecture	Analysis	Browser Modified	Security assumption
Obfuscated-JS [65]	Client side	Offline	No	N/A
SpyProxy [73]	Client side & Network Service	Online	No	Secure virtual machine
Inter-Module [98]	Client side	Online	Yes	Secure browser and OS
DBD-JS [22]	Client side	Online	Yes	Secure browser
Shellcode [31]	Client side	Online	Yes	Secure browser
Strider-HM [116]	Client side	Offline	No	Secure virtual machine
Cujo [83]	Network Service	Online	No	Secure OS in proxy
BLADE [68]	Client side	Online	Yes	Secure OS
DeWare	Client side	Online	No	Secure OS

2.1.3 Browser Security Solution

Many browser security solutions have been proposed, including secure browsers and browser-as-an-OS [50, 115], securing browser extensions [67], and browser mashup security [114, 125]. Our work fundamentally differs from the secure browser line of research, as our solution is completely independent of the browser without any assumption on its or its components' integrity – yielding a more robust detection technique.

2.2 Quantitative Program Behavior Modeling

Anomaly detection, in a big picture refers to the problem of finding patterns in data that do not conform to expected behavior, and the nonconforming patterns are often considered as anomalies [19]. There are many applications of anomaly detection techniques in different areas, such as intrusion detection [25], fraud detection [32], medical and public health anomaly detection [118], and many others. In intrusion detection (a field pioneered by Denning [25]), is further classified into host-based [37] and network-based [3] intrusion detections. Network anomaly detection were studied using data mining or supervised learning techniques, e.g., clustering [41], Markov chains [99]. Sanitization was proposed for preparing network training data [23]. Our work mainly focuses on the area of host-based intrusion detection.

To achieve the goal of host-based intrusion detection, one needs to first understand the intended behaviors of programs, and then enforce proper patterns during execution. However, the task is challenging because of the diversity and well-known unpredictability of the program and system behaviors [21]. Based on the observation that many exploitations and attacks (such as memory overflow, backdoor) alter the normal execution of a program, a good way for detection is to model, monitor and enforce the control flow of a program during execution. This confines the the behavior of a running program to a pre-constructed normal model.

Following the taxonomy in [49], control-flow anomaly-detection solutions can be categorized based on the flow-sensitive property (i.e., the ability to analyze the order of statement executions) or the orthogonal context-sensitive property (i.e., the ability to distinguish calling context at run-time). How the models are constructed, through program analysis or learning, further differentiates them.

2.2.1 Learning-Based or Hybrid Flow-Sensitive Models

Automaton-based models [66, 91] and HMM-based models [44, 117, 124] are flow-sensitive anomaly detection models. With a sufficient large n , n -gram models (e.g., [52]) is also flow-sensitive. [119] pointed out three suitability criteria of n -gram datasets that one can use to assess the suitability of the dataset for a specific learning scheme. The authors experimentally

demonstrate the substantial differences in classification results of different learning methods on datasets. The execution-graph model in [42] was built through learning runtime program execution patterns (namely return addresses on the call stack associated with system calls) and leveraging the inductive property in call sequences. A hybrid pushdown automaton model was presented in [66], where researchers refined the basic statically generated model with program traces in order to cover new transitions associated with run-time properties, such as exception handlers and dynamic libraries. Improvements in classification accuracy and convergence were reported.

Probabilistic data mining techniques were demonstrated for analyzing network intrusions in [63]. The first probabilistic learning work for program behavior modeling was presented by Warrender *et al.* [117] using a hidden Markov model for classification system call segments, which we extensively compare with throughout this work.

The initial parameters of a hidden Markov model are usually randomly chosen, which is inadequate. Authors in [124] systematically investigated the impact of the number of hidden states on classification performance. For certain case, some prior knowledge is also used in HMM initialization. In [117], for the HMM model of *lpr*, the length of document printed has huge impact on the length of read and write pairs in the call traces, which takes a great deal of training time to model with a random start. To overcome that issue, the authors assigned large probabilities to the desired transitions and output system calls for two states representing read and write. [43] uses one HMM to measure the behavior distance of two programs while taking the same input, so that when one is compromised, it can be observed with increased distance. During training and testing, two system call sequences from two programs are aligned. Also, instead of using single system call as emission symbol, a pair of phrases (short sequence of system calls) are used with one from each call sequence. The probability of HMM producing this sequence of phrase pairs is used to measure the distance of the two comparing call sequences. This is a black-box solution which does not use source code or static analysis of binary. When constructing initial HMM, they find that proper choice of N (the number of hidden states) is important and they set N 1.0 to 1.2 times the length of the longest training sequence as a guideline. The choice made by the both [43, 117] shows the necessity of appropriate model initialization.

2.2.2 Program Analysis-Based Flow-Sensitive Models

Instead of learning the automaton model from program traces as in [91], one can construct a similar flow-sensitive automaton by statically analyzing the source code. These statically constructed flow-sensitive models were first demonstrated by Wagner and Dean (non-deterministic finite automaton (NFA) or *callgraph* model in [112]) and later improved by others (e.g., inline automaton model (IAM) in [49]). Dyck model [47] described how flow-sensitive and context-insensitive NFA can enjoy context sensitivity (more below).

2.2.3 Techniques Improving Context Sensitivity

Context sensitivity refers to the ability to recognize different calling context associated with a call, when collecting program traces (for training or for monitoring). There is a tradeoff between the context sensitivity and run-time overhead. For example, as shown in [112] building a context-sensitive push-down automaton (PDA) (in their *abstract stack* model) has prohibitive run-time costs. As pointed out by [49], context sensitivity does not imply flow sensitivity, and vice versa.

Using program counter [91] or call stack information (e.g., dynamically constructed in Vt-Path [34] or statically constructed and more precise in VPStatic [33]) to distinguish calling context have been shown efficient in practice. [46] proposed several techniques for improving context sensitivity of NFA, some of which require program instrumentation such as renaming system calls in order to distinguish different invocations of the same functions. The work also reduced the overhead for PDA. Dyck model [47] inserted code that links the entry and exist of a target function with its call sites, disambiguating call sites, which achieves context sensitivity.

Our STILO-basic model is flow-sensitive, and the extended STILO-context model is not only flow-sensitive but also context-sensitive. The context information used in our model is the caller function of each system or library call, which can be obtained efficiently both at static analysis and runtime monitoring. Besides, as a HMM-based model, STILO has a manageable size controlled at the time of model initialization with statically extracted call transition information.

A comparison of related program behavior modeling work is in Table 2.2.

2.2.4 Consistency in Control Transfer

e-NexSh is an efficient runtime validation system that provides call-stack validation that ensures the consistency in the call site and target site memory addresses for libc call and system call invocations [58]. e-NexSh has high compatibility, as it operates mostly in the kernel space and does not require any modification to application code.

A binary transformation technique was proposed by Abadi *et al.* to achieve control-flow integrity (CFI) [1]. Through modifying source and destination instructions associated with control-flow transfers, it embeds control-flow policies within the binary to be enforced at runtime. Static analysis was used to reduce CFI's overhead in [126]. [127] improved the method by allocating a memory region dedicated to enforcing the targets of indirect control transfers. It brings 2- to 5-fold improvement in the run-time performance. Zhang and Sekar presented static analysis based methods and instrumentation to enforce the CFI property on commercial off-the-shelf binaries [128]. Total-CFI is a framework for system-wide run-time control-flow integrity enforcement [79] built on a software emulator. In comparison to CFI

Paper	Static /Dynamic	Training Data	Test Data	Granularity	Test Apps	False Positive	False Negative	Other
[124]	dynamic FC-HMM LR-HMM	179 traces 846,537 system calls	53 traces 17,309 system calls	system call sequences at different lengths	ps,login, named,sendmail	[0.05, 0.2]	[0, 0.444]	
[117]	dynamic stide	5,264 normal traces	2,177 intrusion traces 75,782 normal traces	system call sequences	lpr,named, xlock,login, ps,inetc, stide,sendmail	$\approx [0, 0.001]$	$\approx [0, 0.5]$	
	dynamic t-stide			system call sequences		$\approx [0, 0.001]$	$\approx [0, 0.5]$	
	dynamic ripper			system call sequences		$\approx [0, 0.001]$	$\approx [0, 0.9]$	
	dynamic HMM			per system call		$\approx [0, 0.001]$	$\approx [0, 0.2]$	
[91]	dynamic	thousands to millions system calls	millions system calls	per system call	nfs,ftp, http servers	$[10^{-6}, 10^{-1}]$		
[34]	dynamic VtPath	millions system calls	millions system calls	per system call	ftpd	$[10^{-6}, 10^{-1}]$		
[60]	dynamic	300 malware	263 malware	per malware sample	malware families: Allaple,Bagle, Mytob,Agent, Netsky,Mydoom		0.36	
[16]	dynamic	[1.75, 4.15] $\times 10^6$ system calls	[3.10, 14.74] $\times 10^6$ system calls	per system call	httpd,sshd	$[0.02, 64.12] \times 10^{-5}$		
[62]	dynamic		MIT Lincoln Labs dataset 127, 637 system calls	per system call	eject,fdformat, ffbconfig,ps,ftpd, sendmail,telnetd	3.06×10^{-4}	0	
			Controlled environment 51, 459 system calls		wuftp,apache, openssh, sendmail, linuxconf	2.91×10^{-4}	0	
			Real-world environment 496, 255 system calls		dhcpcd,imapd, qmail	3.02×10^{-5}		
[112]	static			per system call	finger,qpopper, procmail,sendmail			Branch factor and overhead analysis
[46]	static			per system call	entropy,gzip, randoml,finger, procmail			Branch factor and overhead analysis
[47]	static Dyck			per system call	procmail,gzip, eject,fdformat, ps,cat			Branch factor and overhead analysis
[49]	static IAM			per system call	cat,htzipd, httpd,gnatsd			Branch factor and overhead analysis

Table 2.2: Comparison of different host-based anomaly detection approaches with program behavior modeling.

techniques, our monitoring system is focused on the call-making portion of the control flow. We do not require any binary transformation or software emulator. Most CFI systems assume limited dynamic code behaviors ¹; this assumption is not necessary in STILO because of our trace-based learning component. Unlike STILO, CFI is not designed to offer any probabilistic behavior analysis.

The authors in [92] pointed out through formal analysis that a behavior model on control-flow blocks (as opposed to calls) yield more context-sensitive detection. Monitoring control-block transfers likely incurs substantial run-time overhead.

2.2.5 Data Flow Modeling

Together with control flow, data flow has also been used for anomaly detection. [62] analyzes the arguments of system calls and explores several different metrics for anomaly detection such as the distribution of string lengths and characters. [75] applies multiple detection models to system call arguments to evaluate them from different perspectives, and also introduces a method to combine different results to get an aggregated anomaly score. In [16], the authors developed an approach to learn the data flow behaviors of a program. It learns the temporal properties involving the arguments of different system calls and their relations. The work in [60] described a def-use based ² data-dependence graph at the system-call level for modeling malware behaviors.

In many cases, data flow modeling can work as another layer, and can help achieve better model accuracy together with control flow modeling.

2.2.6 Mimicry Attack

As first discussed in [113], the assumption of a mimicry attack is that the detection model or IDS is known, so the attacker can prepare a system call sequence that matches the detection model but also does malicious activity. This kind of attack will be hard for any IDS to handle. [48, 61, 78] further propose ways to help automate the mimicry attack generation.

Although HMM-based model does not directly detect this kind of attack, [44] is able to perform well on a set of best-estimated mimicry attacks by comparing the behaviors of two programs on same input using also HMM. The way mimicry attack being estimated in the paper [43] is very specific to the way HMM is used in their case, where one model is used to monitor two programs' system call behaviors on same input. As is mentioned by the authors, the input which compromises one program usually induces an error on the other program. This setting means that half of the mimicry sequence pair (the sequence from the

¹E.g., self-modifying code, runtime code generation, and the unanticipated dynamic loading of code [1].

²Definition-and-use based.

non-compromised program) is already abnormal, so when generating a mimicry attack, the attack sequence from the compromised program not only need to look like the abnormal sequence from non-compromised program, but also need to conform to the HMM to get high probability for the sequence pair, which is hard to achieve and as a result makes the detection easier. The main obstacle here for the attacker is that he/she can not directly affect the other program that is running in parallel with the compromised one.

In general, it's hard to detect a mimicry attack which is crafted carefully based on the detection model, but on the other hand, it's a very difficult process for the attackers as well. Mimicry attack is a relatively complex process that usually requires a lot of precise hand crafting and needs a long call sequence to evade detection, which itself is an anomalous observation. Our proposed approach does not directly defends mimicry attack, but makes it harder by having a more accurate detection model.

Chapter 3

Drive-by Download Attack Detection

In this chapter we present the details of our approach for the detection of infection onset, more specifically the detection of drive-by download attack¹. This chapter is organized as follows. We give our models in Section 3.1. An overview of our design of DeWare is in Section 3.2. Our *DeWare* implementation in Windows operating system is presented in Section 3.3. Experimental evaluation is described in Section 3.4.

3.1 Models And Definitions

Security and attack models:

We assume that the browser and its components are not secure and may have software vulnerabilities. The operating system is assumed trustworthy and secure, and thus the kernel-level monitoring of system events and user inputs yields trusted information. The integrity of file systems defined in our model refers to the enforcement of user-intended or user-authorized file-system activities; the detection and prevention of malware-initiated tampering.

Our work targets *application-level malware*. We assume that malware makes persistent changes to the target host's file system or attempts to execute its code by creating a new process, which are commonly observed behaviors at the onset of infection. Similar assumptions were also used in Strider HoneyMonkeys [116].

We do not consider social engineering attacks in this work. For example, a user may be tricked to click on Web links or links in email messages that result in the download of malicious executables. (The recent **Hydra**q malware starts with a personalized spam message with an embedded link to attacker's website [20].)

¹The content of this chapter is mainly based on our NSS work [123]

Intuitively, DeWare detects the onset of infection by detecting anomalies on a host. Given a system event e , we aim to infer the *provenance* of the event by constructing a causal-relationship map that identifies another event that directly or indirectly triggers e .

In our model, we define two types of events: *user action* and *system event*. User actions include keyboard inputs and mouse clicks. There are two approaches to observe user events – one is at the kernel level through the placement of hooks and listeners, and the other is at application level by interfacing with the application that consumes the user actions. For example, at the kernel level the information related to a user’s mouse click may include coordinates of the click, ID of the process receiving the input. In comparison, semantic information associated with the user actions is obtained at the application level, such as the URLs of the links that the user clicks on. System events are any kernel-level transactions such as file system events, network events (outbound and incoming), process events. In this work, for detecting drive-by download we focus on two system events, namely file creation and process creation. Given observed user’s input events and file and process creation events, we compute their causal relations based on pre-defined rules, and to identify suspicious system events that cannot be attributed to any user action.

DeWare runs at the background. Whenever Alice starts an application such as the browser, all her actions and corresponding system events are monitored. A normal system event should be correlated to Alice’s inputs. When the browser is compromised by a maliciously manipulated website, the stealthy events such as executable downloading and execution are detected. Our inference approach involves the comparison of attributes of events including timestamps, file types, process IDs, and semantic information. (more description in Section 3.2.4).

In this work, the causal relation among events is inferred or estimated, thus is not provably-assured. Cryptographic provable provenance has been realized within the operating-system environment for enforcing keystroke authenticity and traffic inspection [104].

Our work belongs to a broader category of research on *human-behavior driven malware detection*. We define that the key research problems in human-behavior driven malware detection include *i)* how to select humans’ characteristic behavior features, *ii)* how to prevent malware forgery, and *iii)* how to make security solutions nonintrusive and transparent to users. Anti-forgery techniques have been proposed with the help of tamper-resistant cryptographic hardware (namely TPM) [51, 104]. Researchers have investigated keystroke dynamics [105], Internet chat behaviors [45] for anomaly-detection purposes. For *iii)* Nonintrusiveness requires any (host-based) security solution to produce *minimal* disruption and annoyance to users.

Security solutions that rely on constant user interactions (for example, through pop-up windows as in [122]) may distract users and cause usability issues. Besides, feedback supplied by users may not be accurate and reliable. Our detection in DeWare collects and utilizes the *existing* human computer interactions, in particular user inputs to applications, for security detection. The tool does not require user to perform additional tasks, thus is easy to use.

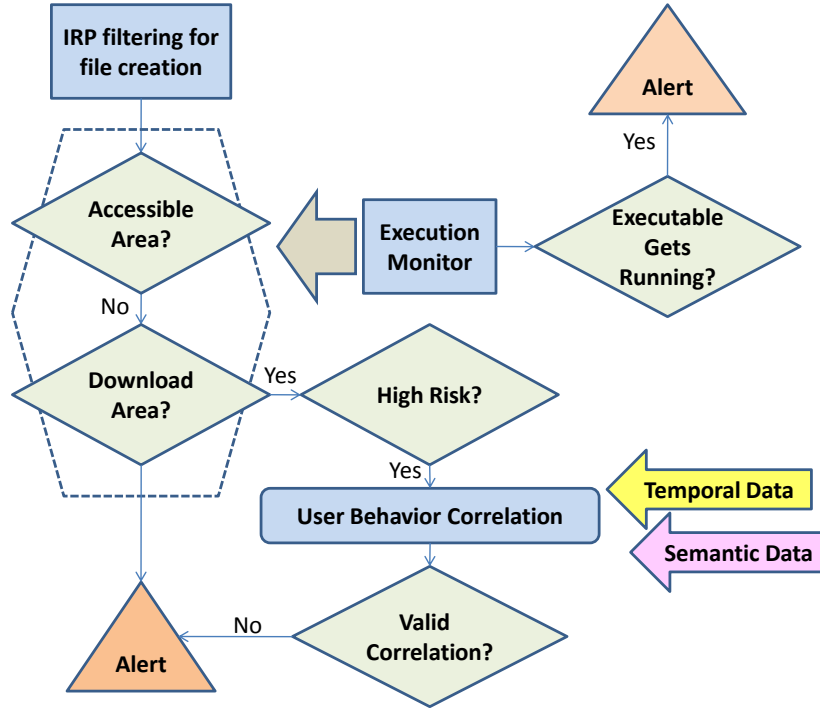


Figure 3.1: Schematic drawing of the work flow for detecting the infection onset on a host based on analyzing user-behaviors and file system/process properties.

3.2 DeWare Design and Its Components

The purpose of DeWare is to detect unauthorized system activities, in particular file and process creations on a host. We consider the system events directly caused by the owner (human user) of the computer legitimate. Malware download and execution are unauthorized. We realize DeWare by instrumenting kernel-level sensors (i.e., kernel modules for collecting and monitoring system events), and identifying legitimate events that are attributed to user actions. Therefore, file creation or process creation that cannot be attributed to relevant user actions is likely to be suspicious and caused by malware infection. DeWare is capable of host-wide monitoring that is application independent. System events that we focus on in this work are the file creation and process creation – either of these two characteristics are often observed at the onset of malware infection. Similar assumption was previously used by others as well [68, 116]. We discuss the limitations of DeWare against more sophisticated malware in Section 3.2.5.

One technical challenge is the fact that many applications (such as the browser) automatically fetch and create files that make persistent changes to the file systems. These downloads may be viewed as indirectly caused by the user. If not carefully identified, these events may create

false alarms. Our solution is to enforce fine-grained access policies on applications to control and confine their access to the file systems. A temporary file automatically downloaded by an application is considered legitimate *only if* it appears in specified areas and is not executed. Violations of the policies are likely caused by malware infection and are reported. The creation of these files may be due to the user visiting a website, i.e., *indirectly caused* by user’s actions. We describe it in more details in Section 3.2.1 and 3.2.2.

This access control framework allows policies to be written and enforced for specifying applications’ privileges of file creation and execution, and significantly reduces our workload of monitoring and analysis.

Users’ intention is embodied by their mouse and keyboard inputs, information of which such as timestamp, corresponding PIDs, and content can be logged at the kernel level or at the application level. Both approaches have different security assumptions and yield user-input information with different granularity. There are pros and cons associated with both methods, which are discussed more in Section 3.2.3. Given the observed user activities and system events, we correlate the two data streams, according to rules on their attributes, e.g., empirically-defined time intervals and process IDs. In our security model, the user inputs obtained are trustworthy, i.e., they are not forged by malware. This assumption can be eliminated if hardware-based attestation (e.g., with trusted platform module) is enabled as described in [104].

Therefore, DeWare is to classify observed system events into one of the three classes: **i)** user downloads, **ii)** application downloads, and **iii)** malware downloads. DeWare includes three main components: *file-system access monitor*, *input recorder*, and *execution monitor*. The work flow in DeWare is shown in Figure 3.1.

- A file-system access monitor is for controlling the access of processes to write to the file system. The policies are defined and enforced through intercepting file-creation related system calls.
- An input recorder is for collecting and interpreting user behavior, in particular user actions regarding downloading files via the browser. We investigate the recording of user actions at both kernel-level and browser-level.
- An execution monitor is for inspecting a portion of the file system for illegal process creation. This component runs with pre-defined policies; it together with the file-system access monitor ensures unauthorized download-and-execute cases are detected at real-time.

3.2.1 File-System Monitor: Confining Applications’ Downloads

Content automatically downloaded by applications is benign, provided that the application is not comprised. Because our security model does not assume the trustworthiness of ap-

plications, it is necessary to examine *all* downloads on the host. However, capturing all file-creation events related to all processes generates an overwhelmingly large number of records. For example, in a study that we performed, a user indirectly triggers 482 file creations in **Temporary Internet Files** folder and 47 in **Cookies** directory within 30 minutes. Files created in those folders are usually benign.

We design and implement a framework that allows us to specify policies to limit applications' access to portion of the file system. The access control framework is to reduce white noise due to application triggered downloads. We define the *accessible area* for an application in the file systems.

Definition 1 *Accessible area of an application is a set of pre-defined directories on the file system where the application is allowed to write to files. The application is not allowed to create or write to files outside its accessible area.*

For example, Temporary Internet Files folder is modifiable by Internet Explorer, whereas system folders are not. The directories to include in the accessible area can be learned based on the software's behaviors, which significantly reduces the area to monitor for file creation, and the number of events to record. With this file-system monitor, malware is confined to the accessible area. Moving executables out of the accessible area is also forbidden.

User download does not need to follow the same requirements. However, the user is not allowed to download files anywhere on the file system, but to a specified *downloadable area*, which is described in Section 3.2.5. With the aforementioned file-system monitor, an attacker may download malware executables to the accessible area of a rogue application, e.g., to the temp folder via a compromised browser. Thus, file-system monitoring alone is not sufficient for detecting drive-by download. We solve the problem by monitoring execution patterns in these download areas, which is presented in the next section.

3.2.2 Real-Time Execution Monitoring on Host

Although dormant malware (downloaded but not executed) does not pose immediate threat to the host, it may get executed later on. In our experiments with real-world malware, some malware starts itself after a reboot. In DeWare, execution monitor is to prevent malware stored at accessible area from being run. Execution monitor inspects the accessible areas granted to applications.

For a newly created process, we collect information including process ID, image file name, parent process ID, and timestamp. Once a new process is created, the execution monitor is notified. It verifies that the image file is not in the accessible area of any application. As our security model assumes that the kernel is not compromised, the process information collected is complete and trustworthy. We do not consider the existence of rootkits that hide

their presence in the process table. As we aim to detect the onset of the infection on an otherwise clean host, this assumption is valid.

3.2.3 Input Recorder

There are several approaches for obtaining user intention on a computer: *i*) directly asking the user (e.g., through a pop-up window) [122], *ii*) analyzing user’s transaction history and extracting patterns [93], or *iii*) recording the events entered through keyboard or mouse devices [104]. Each method has pros and cons. For example, pop-up window may be intrusive to user, but is easy to implement. In DeWare, we opt for the third approach.

There are two levels of user-input monitoring: kernel level and application level. Kernel-level input logging records user inputs at their origin, as it is to collect events triggered by activities on external input devices via device drivers. It intercepts information about user inputs – coordinates, timestamp, and content – by placing listeners or hooks in the kernel. The inputs are consumed by the current foreground process, whose ID can be identified. Kernel-level input logging is application-independent, which makes the method general.

User inputs obtained in the kernel space are assumed to be trusted in our model. This assumption can be relaxed if additional assurance measures such as cryptographic provenance verification on keystroke/mouse are deployed [104], which prevents malware from forging user actions.

One can also record user inputs from the application. This approach requires writing a plug-in specific to the application. It also assumes the trustworthiness of the extension to ensure the data integrity. The obvious advantage of application-level input logging is the ability to *semantically* interpret certain input events, such as the URL that a mouse clicks on. This URL is application-specific data and cannot be easily obtained outside the browser. At the kernel level, a user’s mouse click is not associated with this semantic information. In our prototype in Windows, we realize and compare both methods.

3.2.4 Dependency Rules

User event such as a left mouse click has specific *semantic* meanings in the application that consumes the event. More specifically in the context of download event, semantic properties of a mouse-click event include the information about the file that the user intends to download, for example, source URL, file name, type, size, destination directory, as well as temporal information (i.e., timestamps of input events), and properties of the process consuming the input event. Therefore, our inference rules are defined to inspect and compare the file, process, and temporal properties between user actions and system events.

We formalize our dependency rules as follows. Consider a file-creation system event e with

attributes $(fname, fpath, ftype, T, pid, pname)$, where $fname$, $fpath$, $ftype$ are the name, path, and type of the file, respectively, T is the corresponding timestamp, pid and $pname$ are the process ID and name of the application that creates the file, and consider an user-download event e^* with attributes $(fname^*, fpath^*, ftype^*, T^*, pid^*, pname^*, intended_dl_URL, actual_dl_URL)$, where $fname^*$, $fpath^*$, and $ftype^*$ are the name, destination path, type of the file obtained through parsing the user event, T^* is the corresponding timestamp, pid^* and $pname^*$ are the process ID and name of the application that consumes the user event. $intended_dl_URL$ is the URL where user is attempting to download a file, while $actual_dl_URL$ is where the application actually makes the download.

We define the dependency between events e and e^* (i.e., file-creation event e is caused by a user mouse-click event e^*) as the satisfaction of the following rules.

- *Rule 1:* File properties of events match: $fname = fname^*$, $fpath = fpath^*$, and $ftype = ftype^*$. That is, the file to be created should literally match the file that a user intends to download.
- *Rule 2:* Process properties of events match: $pid = pid^*$ and $pname = pname^*$. Both events should be for the same application. Child processes are considered as the same application.
- *Rule 3:* Temporal constraint is satisfied: $0 < T - T^* \leq \tau$. That is, a legitimate file-creation event is required to take place within a short threshold τ after a valid user-input event. Different τ values are experimentally evaluated in our user study in Section 3.4.
- *Rule 4:* User intention is ensured: $intended_dl_URL = actual_dl_URL$. User's download intention should be correctly reflected by application's action. For example, a file creation of `iTunesSetup.exe` to the system can be correlated back to the events of
 - i) user clicks to initiate and confirm the download from
`http://appldnld.apple.com.edgesuite.net`
 - and ii) the browser goes to fetch the executable from
`http://appldnld.apple.com.edgesuite.net/.../iTunesSetup.exe`.
 It means that file should be downloaded from the source URL that user intends.

Additional rules can be introduced to capture and specify contextual properties of the download and file-creation events. We note that temporal-constraint alone does not provide sufficient security protection against infection onset. Consider coincidental malware download as follows.

Definition 2 *We define coincidental download as the download of malware that coincidentally occurs immediately after a user's mouse click, more specifically the time interval between the download event and its immediately preceding mouse-click event is within threshold τ .*

The problem of coincidental download only exists when logging user’s mouse events at the kernel level, which lacks semantic and contextual information of the events. For example, a click on a URL cannot be distinguished from a click on download-dialogue box. This problem is prevented when using application-level input logging as described in Section 3.2.3.

3.2.5 Piggybacking Download and its Detection

Piggyback download defined in [103] is where malware is part of a piece of software downloaded with proper user permission, e.g., spyware bundled with compression software. Preventing piggybacking code from being downloaded to the file system is difficult in our model. Thus, our approach is to detect it when the downloaded malware is being executed. The area that execution monitor inspects needs to be expanded beyond the accessible areas of applications. The execution monitor also inspects the area at which user-authorized downloads are stored, and seeks confirmation from the user if files in that area get executed. We define downloadable area in Definition 3 below.

Definition 3 *Downloadable area of a user is a set of pre-defined directories on the file system to which the user downloads files from the Internet.*

If piggybacking download in this area creates new processes, the execution monitor prompts the user for additional approval. Note that the user interaction is only needed for files with download-and-execute patterns; *not* for each process creation in the system. Thus, the required user interaction is minimal. As stated in Section 3.1, we assume that the host is *not* infected with malware that is capable of eavesdropping on user mouse events (and downloads immediate after the user clicks the mouse). This assumption is valid, as DeWare aims to detect the onset of infection on a clean host.

3.3 Prototype Implementation in Windows

We describe our implementation of DeWare prototype in Windows XP utilizing some existing kernel driver and tools for monitoring system events in Windows. The prototype is easy to deploy and efficient to run. The framework is described in Figure 3.2, which shows how DeWare collects and filters file-system events, monitors processes, collects user inputs, and integrates and analyzes data, respectively.

3.3.1 File-System Monitor and Execution Monitor

Our file-system monitor consists of a kernel-space driver and a user-space component. File-creation events are collected at the kernel level and reported to the user space for filtering.

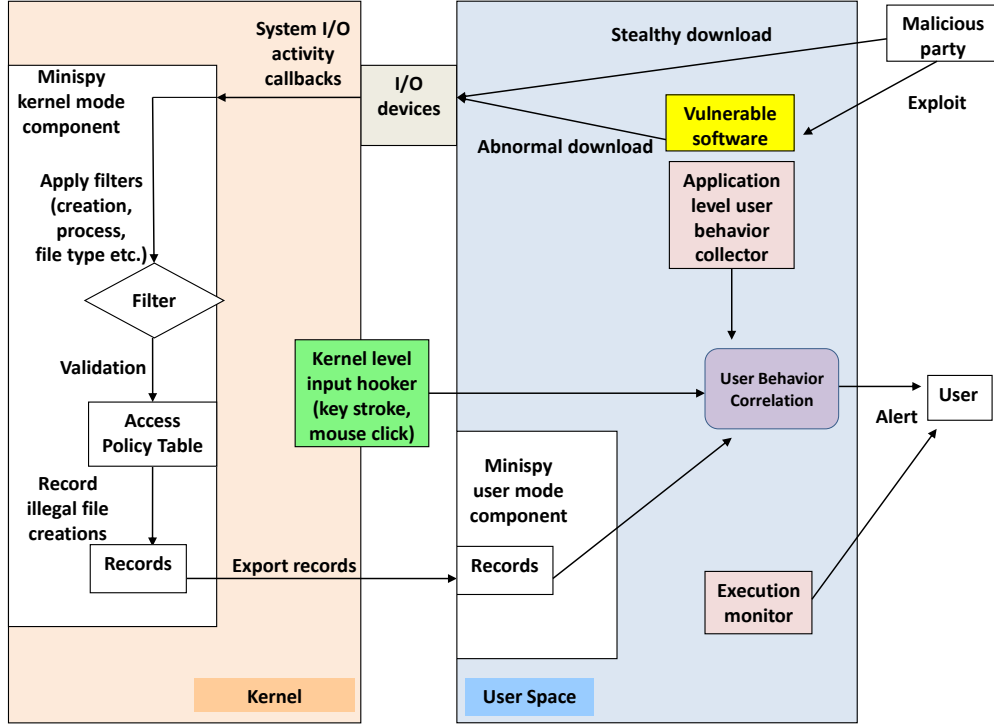


Figure 3.2: The system framework of DeWare implementation.

The kernel-space driver filters all IRP (I/O request packet) calls issued by operating system to the low level I/O devices. We specify the applications to be monitored and focus on the `FILE_CREATE` system call. Other file-related system calls such as `FILE_OPEN` and `FILE_OVERWRITE` may be recorded as well. For intercepting file-related system calls, our prototype utilizes an existing kernel driver (Minispy) for Windows OS that can monitor all system calls involving opening a handle to a file object, including file creation.

We use `minispy`, which is a kernel driver for Windows operating systems, to monitor all local disk partitions except the temporary folders used by browsers. Minispy is a tool which is able to monitor and log any I/O activities happening in the system. It is composed of two components, namely kernel-mode component and user-mode component. The kernel mode component registers callback functions with a file system filter driver. So when this filter driver intercepts requests of I/O operations, via those registered callbacks minispy kernel-mode component is able to record any of those I/O operations. Then a user can request the recorded information, and it is passed to user-mode component, where the recorded information can be directed to different output. In our modified minispy, the requested record information is saved to local file system.

DeWare keeps track of targeted processes as well as their child processes. We organize the directories in both accessible and downloadable areas with the an array data structure for each process. If a file is to be created outside the accessible area of the process, then the event

along with its attributes is reported to the user space. This verification has low overhead due to small search array sizes. In the downloadable area of user, high-risk files as defined by [74] in that area are examined in our dependency analysis. Certain low-risk downloads (e.g., TEXT files) can be safely ignored, but we monitor the area for unauthorized execution.

If a file creation operation fails to pass all those filters and temporal evaluation, it triggers the alert application to warn the user of this file creation event. In our prototype, we do not block or suspect the (suspicious) system call.

Details of how we obtain file creation events are described as follows. The Windows operating system sends most of its I/O requests to drivers using I/O request package, which is a persistent data structure and contains all the information a driver needs to know in order to respond to an I/O request. By registering callback functions to one of the IRP function codes, `IRP_MJ_CREATE`, Minispy is able to monitor all the events where system is requesting to open a handle to a file object or device object. By further checking the `FLT_PARAMETERS` for `IRP_MJ_CREATE` call, we are able to decide whether this I/O operation is for `FILE_CREATE`, or `FILE_OPEN`, or `FILE_OVERWRITE`, etc.

Our execution monitor is implemented based on **PsTools** suite, which is an existing collection of command-line utilities to manage processes in Windows. The tool leverages the security-monitoring features provided by Windows OS (XP and higher), which comes with security settings for monitoring the local host, and among those the **AuditPolicy** is able to track all the processes. The execution monitor records all the process **start** and **exit** events, and applies policies to inspect them for violations. A sample log entry collected is shown as follows.

```
A new process has been created:
New Process ID: 3444
Image File Name: C:\WINDOWS\system32\cmd.exe
Creator Process ID: 1396
User Name: Administrator
Domain: XMENXK-50C6105A
Logon ID: (0x0,0xEB4D)
```

3.3.2 Collect and Correlate User Behavior Information

DeWare implements two independent mechanisms for collecting user inputs: one at kernel level and another within the Firefox browser. The kernel-level logging records user inputs through hooks `SetWindowsHookex` provided by Windows OS. The choice of proper temporal parameters is discussed and evaluated in Section 3.4. We also develop a Firefox extension (based on **tlogger**) which is capable of recording users' clicks that correspond to downloading activities. This extension provides semantic properties of user input as shown in Table 3.1. We note that if a user clicks on buttons within a browser plug-in (e.g., Adobe Reader) to

Table 3.1: User inputs recorded through our Firefox extension.

Download Scenarios	Recorded
Clicking on a link	Yes
Typing a URL into address bar	Yes
Using “Save Target As...” button	Yes
Download initiated by page redirect	Yes
Download from embedded plugin	No

download, then the extension is unable to record the event. With the gathered semantic properties, the accuracy of dependency inference can be significantly boosted.

Our extension captures user’s activities on a web page and also the information from the common download-dialog window, which asks the user for file-download confirmation. We retrieve the download-event related attributes such as file name, source, and file type, which is defined in Firefox the user interface for download in the file `unknownContentType.xul`. The download-related attributes that we need to collected are specified in new `xul` (XML User Interface Language) files: `dbdlogger.xul` and `savebox.xul`, which are included in the Firefox extension and specified in the manifest file as:

```
content dbdlogger content/
overlay chrome://browser/content/browser.xul
        chrome://dbdlogger/content/dbdlogger.xul
overlay chrome://mozapps/content/downloads/unknownContentType.xul
        chrome://dbdlogger/content/savebox.xul
skin dbdlogger classic/1.0 skin/
```

Overlays [77] are XUL files that are used to describe extra content in the UI components for Firefox. When plug-ins, extensions, or other applications provide new UI elements to the browser, these new elements should be defined in overlay files, so as to work together with the present XUL files which construct the basic UI interface. The added overlay contains codes which extend the original Firefox UI to add and realize certain functions an extension needs. Within the XUL files which specify our additional extension functions, we include JavaScript to listen for user’s behavior such as click on the OK button. Once the user confirms her download in the dialog window, our extension records the relevant attributes regarding this download event into a log file. We also add JavaScript for changes of download states, such as `DOWNLOAD_DOWNLOADING`, `DOWNLOAD_FINISHED`. A snippet of our JavaScript code for parsing the save-file dialog is shown in Figure 3.3.

Figure 3.3.2 gives an example of how correlation is established among different download phases to authenticate a normal file creation, starting from a user click to a finished download.

In our prototype, the dependency analysis is performed off-line, after data is collected. We also implement a real-time system based on temporal correlation. Our solution does not

create any new privacy vulnerability. DeWare is a stand-alone host-based solution that does not export the collected data out of the user’s computer. Collected data is erased after the analysis and does not need to be stored for a long term.

The detailed analysis algorithm in pseudo-code is shown in the Figure 3.5.

Limitations DeWare is capable of detecting a wide spectrum of syndromes associated with infection onset. Our detection assumes that malware either makes persistent changes to the disk or creates its own new process. Thus, DeWare cannot detect the infection onset where code or dynamic loadable library (DLL) is injected into the memory of a legitimate process [95, 96, 97]. This type of in-memory injection does not need to touch the hard disk and the malicious code can run in the context of the compromised process without the creation of a new process.

An attacker could circumvent our detection by injecting fake inputs, or clickjacking, and the purpose is to bypass correlation and to authenticate illegal downloads. This type of injections could be identified with the help of a cryptographic hardware attester, such as TPM [104].

3.4 Experimental Evaluation

We carry out extensive experiments to evaluate the effectiveness and usability of our solution. 21 users are recruited in a user study to collect real-world user download behavior data. We also use DeWare to evaluate a large number of both legitimate and malicious websites for testing its accuracy.

3.4.1 Detecting Known DBD Exploits and Real-World Malicious URLs

We test DeWare against real-world websites with drive-by download exploits [69, 70]. The testing system is Windows XP Version 5.1 (Service Pack 2) installed within VMware 7.0. Internet Explorer Version 6.0.2900.2180 is used as the victim application. We take a snapshot of the clean system and then test DeWare with malicious websites. We give each URL one minute to load and launch any attack and revert the system to the initial clean state before each new test. During a two-week period, we successfully detected 84 unique domains with drive-by download exploits. Here are the observations.

- The malicious websites typically download executables and .dll files onto the victim’s host, and then try to get the executables running. The entire procedure is surreptitious. In some cases, after reboot the browser is automatically loaded and re-directed to pornography websites.

- There are several popular exploit kits, such as Phoenix exploit kit and Eleonore exploits pack, which are widely used by many malicious websites. They target at multiple software vulnerabilities including Flash, PDF, Java and browser.
- There are websites who track the incoming requests. In that case, the first visit triggers the exploit, while during the second visit no exploit is observed or the web server refuses the connection.
- Some exploits attempt to download executables to directories such as `Documents and Settings\Administrator\Local Settings\Temp\` to avoid detection, which can be detected by DeWare.

We also produce several known drive-by-download exploits in a lab environment shown in the following. DeWare can successfully detect executables downloaded as a result of a successful exploit.

- Heap Feng Shui attack [100], which targets MS06-067.
- HTML Object Memory Corruption Vulnerability [53];
- Superbuddy exploits through AOL activeX control [108];
- Adobe Flash player remote-code execution [36].
- Microsoft Data Access Component API misuse [30];
- DBD exploiting IE 7 XML library [54].

An evaluation of other off-the-shelf security products' effectiveness against IE 7 XML DBD attack [54] is shown in Table 3.2.

3.4.2 User Study

21 users participated in our user study – all of them are graduate or undergraduate students from a university. Each user is asked to surf the web for 30 minutes and download at least 10 files of her choice. In Figure 3.6, we give the histogram of the observed intervals between a user's mouse-click event and the corresponding file-creation event. For this analysis, the two types of events are correlated manually by the authors. User's click events are recorded at the kernel level through a mouse hook to the input device driver, and the timestamps for file-creation events are extracted from the intercepted system calls. The majority of user-triggered download have a short delay within 80 milliseconds.

The false positives based on the temporal correlation (i.e., comparing timestamps of events) are reported in Figure 3.7(a). False positives in our user study may come from two sources.

Table 3.2: Comparison of off-the-shelf security products' effectiveness against IE 7 XML DBD attack [54].

Product	Driver Engine Version	Definition	Reaction
360 Safe-guard	v3.0	2008-6-16	No detection
	360 v6.0.1	2008-6-16	No detection
	360 v6.0.1	2009-10-27	No detection
	360 v6.0.2	2009-10-14	Detected heap spray attack, shutdown iexplorer.exe
Zonealarm Pro	7.0.483	Anti-spyware engine 5.0.189	Captured .exe trying to access Internet; user clicked "Deny"; but .exe was still downloaded
	8.0.400	Anti-spyware engine 5.0.209	
	9.1.008	Anti-spyware engine 9.1.008	
Trend Micro Internet Security Pro	v8.952	Pattern version 6.289	No detection
		Pattern version 6.587.50 2009-10-29	No detection
Microsoft Security Essentials		Virus Definition 1.69.825, Spyware Definition 1.69.825 2009-11-11	Detected threats; user clicked "Clean the threat"; but .exe files were still downloaded
AVG Internet Security	v8.5.423	Virus DB 270.14.20 2009-03	Threats detected; page blocked
	v9.0.707	Virus DB 270.14.68/2507 2009-11-16	Detected threats; page blocked
McAfee	v8.1 Build 8.1.175	VirusScan v12.1, 2009-10-28, Personal Firewall v9.1 2009-10-28, Anti-Spam v9.1 2009-10-29	Buffer overflow attack captured
Kaspersky Internet Security	v7.0	Database 2007-10-1	Detect suspicious "Data Execution"

i) File creation from user download in the downloadable area of the user that has a high risk extension, but is not within the required threshold, and *ii*) (legitimate) file creation by the browser that is not in the accessible area. During the user study, we have six such violations, e.g., `\Program Files\NOS\bin\getPlusPlus_Adoobe.exe`, which all result from a single installation of *getPlus* (a download manager from NOS). *getplus* is piggybacking downloaded with Adobe Reader.

The results show that the number of false alarms that temporal-comparison based DeWare generates in our user study is small ($< 1\%$) compared to the total number of file creations. A larger threshold leads to a lower false positive rate, but at the same time it also increases the likelihood of false negatives as we show earlier. Given our results, we recommend a threshold within 80-100 milliseconds. The false positives are further reduced with the application-assisted semantic comparison.

For eight participants in our user study, we also collected application-level inputs using the Firefox extension described in Section 3.3. It records users' mouse-click activities within webpages and on download-dialog windows, and obtains semantic information of the user event such as file name, type, size, source URL, timestamp, and destination directory. Without applying any dependency rules, there are 34 false positives for these eight participants in our dataset. With the temporal constraint (*Rule 3* in Section 3.2.4) and process constraint (*Rule 2*), our analysis generates 16 false positives. The threshold τ is set to 100 milliseconds. With all rules including the comparison with semantic properties of user-input events, we further reduce the false positives from 16 to 6. Figure 3.7(b) shows that the semantics of user actions helps reduce the false positive rate. These remaining six false positives are due to JavaScript files that are bundled together with a webpage a user downloaded.

False alarm evaluation with legitimate websites. Given the pre-defined accessible area (consisting of 18 folders) of Firefox, we test DeWare with legitimate websites to see if there are any false alarms. A false alarm may be caused by a website *i*) downloading files outside the accessible area, or *ii*) creating new processes from files stored in the accessible area. We automatically evaluate 2000 websites from [Alexa.com](http://www.Alexa.com) on August 20th, 2010. We give the browser 30 seconds to load a webpage. We have zero false alarm in our experiment. The result indicates that our pre-defined accessible area is sufficient in containing the files that the browser and common plug-ins create. We demonstrate the feasibility of confining a process' access to file system for reducing DeWare's workload of system-call monitoring. We are able to achieve it without the need of modifying how browser operates.

```

<dialog id="unknownContentType"
  ondialogaccept="return logSaveAs(dialog.onOK())">
  <script type="text/javascript">
    function logSaveAs(retval)
    {
      try {
        globals = Cc["@dbdlogger/globals;1"].getService().wrappedJSObject;
        log_write = globals.buildLogFunction();

      } catch (ex) {
        alert("Logwrite Initialization failed: " + ex);
        return retval;
      }

      try {
        filetype = document.getElementById("type").value;
        source = document.getElementById("source").value;
        filename = document.getElementById("location").value;
        log_write("SAVE_AS",
{"filename":filename, "filetype":filetype, "source":source});

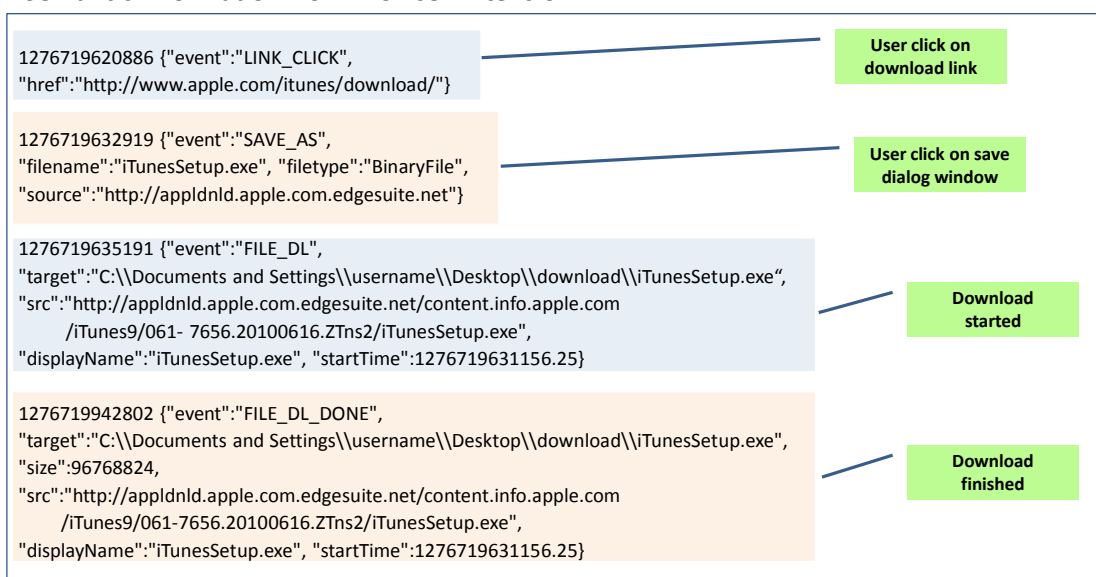
      } catch(ex) {
        alert("Error logging SAVE_AS: " + ex);
        return retval;
      }

      return retval;
    }
  </script>
</dialog>

```

Figure 3.3: JavaScript code for parsing save-file dialog and saving attributes associated with the user's mouse-click action.

Semantic Information from Browser Extension



File Creation Captured by System Logger -- Minispy

IRP	0x00006B89	16:20:34:718	16:20:34:718	d38.234
	\Device\HarddiskVolume2\Documents and Settings\username\Desktop\download\iTunesSetup.exe			

Figure 3.4: An example of semantic correlation.

Inputs: File creation events filtered from I/O request packets: $C^1, C^2, C^3, \dots, C^i, \dots$
 Process creation events from system built-in security auditing: $P^1, P^2, P^3, \dots, P^i, \dots$

Outputs: alert if safety policy violated

#Define

Threshold: T = latency between user input and corresponding file creation

A set of high-risk file extensions: **Hi-risk** = { .bat, .cmd, .exe, .js, etc }

Accessible area: **A-area** = { C:\...\Local Settings\Temp, C:\...\Temporary Internet Files, etc }

Downloadable area: **D-area** = { D:\My Documents\Downloads, etc }

*/*file creation monitor*/*

for each C^i **do**

if ($\text{Path}(C^i) \notin \text{A-area AND Path}(C^i) \notin \text{D-area}$)

 Generate an alert

if ($\text{Path}(C^i) \in \text{D-area AND Extension}(C^i) \in \text{Hi-risk AND !Correlation_Success}(C^i)$)

 Generate an alert

end for

*/*process execution monitor*/*

while (true) **do**

if $\text{Path}(P^i) \in \text{A-area OR Path}(P^i) \in \text{D-area}$

 Generate an alert

end if

end while

Figure 3.5: DeWare's Detection algorithm in pseudo-code

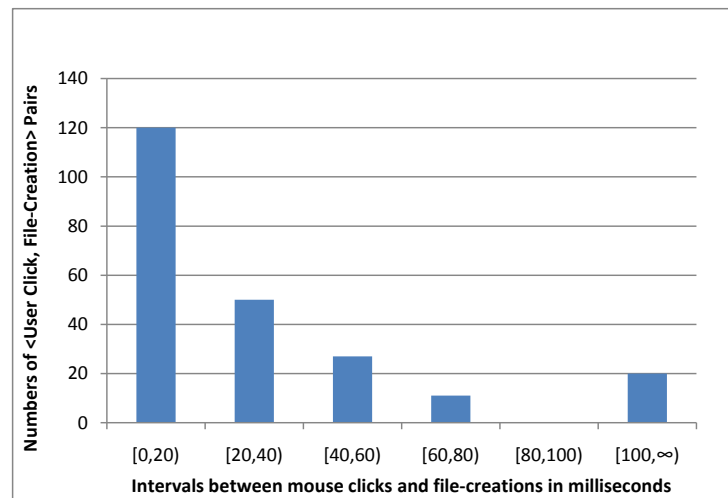
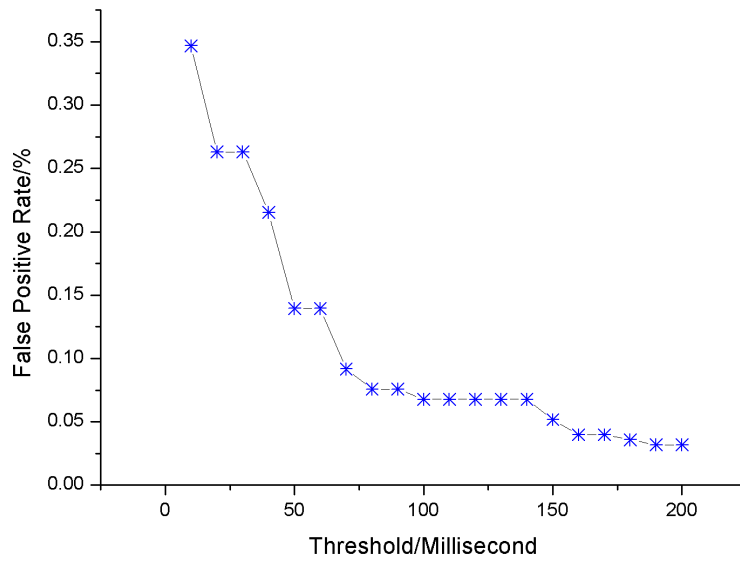
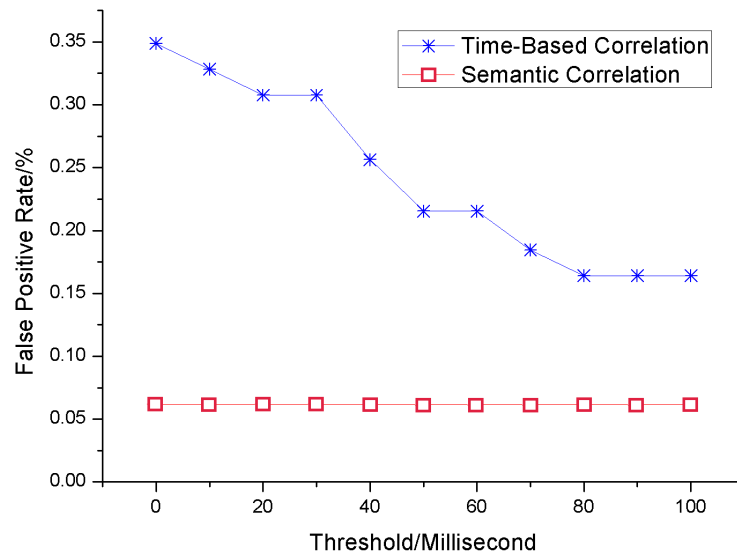


Figure 3.6: Histogram on the interval in milliseconds of the user click events and their corresponding file creation events.



(a)



(b)

Figure 3.7: (a) The averaged false positive rate vs. various threshold values based on a total of 25,092 file-creation events including temporary files by the browser. (b) Reduced false positives with semantic-based comparison between user's mouse-click events and downloaded files.

Chapter 4

Quantitative Modeling of Program Call Sequence Behavior

Our DeWare approach of utilizing user interactions for authenticating system events is promising, especially against drive-by download attacks. However, we also see that, although a common observation, not all intrusions or attacks end up downloading malicious code into the compromised machines. It's possible that the malicious code only stays in the memory of the vulnerable application. Also the availability of user interactions does not apply to many applications such as FTP servers. In this chapter, we present a more general approach which builds quantitative models with increased accuracy, for program behavior modeling, which does not rely on user input behavior. We give an overview of our approach in Section 4.1. Details of PROBABILITY ESTIMATION, AGGREGATION OF CALL TRANSITIONS, and HMM INITIALIZATION operations are given in Section 4.2. Our evaluation is presented in Section 4.3.

4.1 Overview of Our Approach

Our approach aims at building a probabilistic flow-sensitive model to capture the normal control flow of a running program. Following [29], we first define the control-flow graph of a function below.

Definition 4 *The control-flow graph (CFG) of a function is a directed graph, where nodes represent code blocks of consecutive instructions identified by static program analysis, and directed edges between the nodes represent execution control flow, such as conditional branches, and calls and returns. Calls include system calls, library calls or user-defined function calls.*

Consider an edge ($n_1 \rightarrow n_2$) in a control-flow graph, we refer to n_1 as the parent node and n_2 the child node. A CFG node may or may not make a call. E.g., in the simple control-flow

graph shown in Figure 4.3, node 3 calls syscall `write`; node 5 does not make any syscalls. ϵ and ϵ' represent the external call site and return site of $f()$.

Given the CFGs for all functions, a possible but simple approach for building a program behavior model is as follows.

1. Aggregate the control flow graphs of all the functions into a single augmented call graph (ACG), which is the integration of each function's control flow graph and the overall call graph of a program.
2. Perform some form of probability estimation to quantify the call sequence information in the ACG.
3. Use the estimated probabilities to initiate a hidden Markov model for recognition.

However, this above workflow is impractical, because estimating or extracting all call paths in such a large ACG is of exponential complexity in the number of graph nodes. Also the size of the augmented call graph can be very large and a larger model can cause heavy overhead, especially when being used at runtime. Thus we need a way to extract the useful static information from the CFGs and build an efficient anomaly detection model.

Our work provides a probabilistic representation to the control-flow nodes that make system calls or library calls. This new representation enables a more complete coverage of both dynamic and static program behaviors in a hidden Markov model (HMM). A hidden Markov model (HMM) probabilistically represents a Markov process consisting of unobserved interconnected hidden states, which emit observable symbols [82]. Next, we first give a basic introduction about hidden Markov model, which is the base model we are improving upon. Later we present a toy example in Figure 4.1 to illustrate the pros and cons in some program behavioral modeling approaches and our advantages brought by modeling and computing the probabilities for observed call sequences.

4.1.1 Hidden Markov Model

Initially introduced in the late 1960s and early 1970s, hidden Markov model becomes popular because of its good performance in several important practical applications, especially speech machine recognition [82]. As a simpler form, a Markov model has states that are directly visible to observer and the transition probabilities among states are the only parameters. In comparison, a hidden Markov model's states are not directly visible, but the outputs or emissions of the hidden states are visible. As a result, the parameters for a HMM includes not only the transition probabilities among states, but also the emission probabilities for different observations out of each hidden state. The rich mathematical structure makes HMM work well for many applications.

In detail, a HMM is defined by the basic elements including:

- N , the number of hidden states in the HMM. The collection of hidden states is represented as a set as $S = \{S_i\}$, where $1 \leq i \leq N$.
- M , the number of distinct observation symbols for each state.
- $A = \{a_{ij}\}$, the state transition probability distribution. If we denote the state at time t as q_t , we have $a_{ij} = P[q_{t+1} = S_j | q_t = S_i]$, where $1 \leq i, j \leq N$.
- $B = \{b_j(k)\}$, the observation symbol probability distribution in state j , where $1 \leq j \leq N$ and $1 \leq k \leq M$.
- $\pi = \{\pi_i\}$, the initial state distribution. $\pi_i = P[q_1 = S_i]$, and $1 \leq i \leq N$.

The way we use HMM here is mainly for the question: Given the observation sequence $O = O_1 O_2 \dots O_\tau$, and a model $\lambda = (A, B, \pi)$, how to efficiently compute the $P(O|\lambda)$, which is the probability of the observation sequence, given the model? The Forward-Backward Procedure [12, 15] is available for performing this computation. This allows us to use a HMM to monitor new observation sequences.

Another important question is: How to adjust the model parameters λ to maximize $P(O|\lambda)$? This is basically about how to train a HMM to learn about normal data. The fact is that, given a finite set of observation sequence as training data, there is no optimal way of estimating the model parameters. However, we can use an iterative procedure (such as the Baum-Welch method [13, 14, 11]) to find a local maximum. We use these available methods for HMM training and testing.

4.1.2 A Motivating Example

Here we use an example to better present the motivation of our work. Suppose a function or program has three execution paths (P_1, P_2, P_3), where paths P_1 and P_3 are likely to occur during the program execution. Although statically feasible, P_2 has a very low probability to be executed.

For a learning-based approach, program behavior models are constructed based on system traces that are collected when the trustworthy version of the program executes.

- (*Pro*) Can approximate the frequencies of program behavioral patterns (e.g. using HMM as done in [117]).
- (*Con*) Incomplete training set results in false alarms. As shown in Figure 4.1, system call sequences containing the rare-but-statically-feasible path P_2 may be misclassified as abnormal.

Common paths: P_1, P_3

Rare path: P_2

Program-Analysis Approach

Model:

P_1	Possible
P_2	Possible
P_3	Possible

Test traces	Classification result	
$P_1P_3P_3$	✓	All statically feasible on program model
$P_1P_2P_3$	✓	
$P_2P_2P_2$	✓	

Cannot differentiate occurrence frequencies (common vs. rare)

Learning Approach

Training traces: $P_1, P_3, P_3, P_1, P_1, P_3$

(Incomplete training data, not covering rare path P_2)

Test traces	Classification result	
$P_1P_3P_3$	✓	Seen before
$P_1P_2P_3$	✗	P_2 is new
$P_2P_2P_2$	✗	P_2 is new

Cannot recognize new feasible paths not covered in training

Figure 4.1: Illustrations of classification deficiencies in program behavior models that are constructed from static program analysis (left) or program traces (right). Suppose there exist three statically feasible execution paths P_1 , P_2 , and P_3 , among which paths P_1 and P_3 are much more frequent to occur than the rare path P_2 . The paths represent the system call sequences.

For a program-analysis based approach, feasible control flow information is extracted through statically analyzing the code.

- (*Pro*) Can discover all statically feasible execution paths.
- (*Con*) Cannot differentiate the likelihoods of occurrences among feasible paths. As shown in Figure 4.1, a highly unlikely call sequence $P_2P_2P_2$ (an indicator of possible exploits, or abnormal program behaviors) cannot be detected.

Through a new program-analysis based probability forecast, our method makes two improvements. **i)** With a probabilistic representation for call sequences, it differentiates their frequencies of occurrences, improving detection sensitivity. It computes a probability $P(\langle c_1, \dots, c_k \rangle | \lambda)$ for an observed call sequence $\langle c_1, \dots, c_k \rangle$ for a given hidden Markov model λ . A larger probability indicates more likely for the call sequence to occur in normal program execution. It can identify feasible-but-unlikely sequences (e.g., $P_2P_2P_2$). **ii)** Because our HMM model is initialized with statically predicted call-transition information, it has the potential to recognize legitimate new calls as well as new call sequences that do not appear in the training set.

Alternative Design: Consider a straightforward hybrid approach for building a program behavior model, where one may use two independent models – a program analysis model (e.g., [49]) and a quantitative learning model (e.g., [38]) – to classify. This approach utilizing existing techniques is easy to implement. However, how to intelligently reconcile the two votes from the two methods is unclear. If not done properly, a straightforward hybrid approach may suffer from the inherent limitations of both paradigms.

In comparison, our technique organically integrates probabilistic learning with static control flow. Our statically derived control-flow information about paths allows estimation of path frequencies; our dynamic traces refine the behavior model when augmented by this path frequency data and by static paths not seen in the traces.

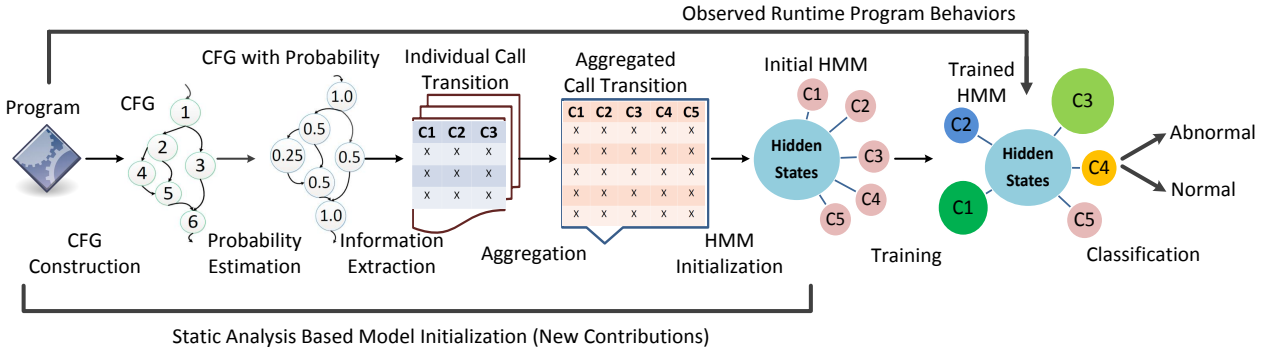


Figure 4.2: A diagram illustrates our anomaly detection workflow.

4.1.3 Design Overview and Workflow

Our program-analysis-guided probabilistic detection has the capability to detect stealthy attack sequences. It has the capability to reason on occurrence likelihoods beyond the binary feasibility prediction. A diagram illustrating our workflow is shown in Figure 4.2. We give an overview of our workflow below.

1. **Probability Forecast:** We extract information from control-flow graphs to statically estimate likelihoods of occurrence for call sequences. This probability forecast is completed through two steps.
 - *Step 1:* One takes a control-flow graph of a function and outputs a *call-transition matrix* for this function (Definition 8). This matrix consists of estimated call transition probabilities, which represent the likelihoods of occurrence for sequences of calls when the function $f()$ is executed.
 - *Step 2:* To obtain the call transitions of the entire program, one aggregates individual transition matrices of functions into one (larger) matrix. The aggregation

of probability values are performed according to the call relations between the caller and callee functions in the call graph.

2. **HMM Init:** This operation takes as an input the call-transition matrix of the program and initializes the parameters of a machine learning model (namely hidden Markov model), including the number of hidden states N , the collection of observation symbols and its number M , emission probability distribution matrix B representing likelihoods of emitting observation symbols by hidden states, transition probability A among hidden states, and the initial probability distribution π for hidden states.
3. **Train and Classify:** Training with normal program traces tunes the parameters of the HMM learner, so that it can recognize dynamic code behaviors. At classification, when given a segment of program traces (in system call or library call), the model computes the probability of the call segment. This probability is the summation over all possible hidden state sequences (using the forward algorithm). The classification decision is made with respect to a pre-defined threshold T on the production probability of a call sequence.

Our model is flow-sensitive, as the Markov model captures the order of execution of statements in the program. Flow sensitivity is important for building high-precision anomaly detection systems. Automaton-based models (e.g., statically constructed [49] or dynamically learned [91]) are also flow-sensitive.

4.1.4 Probability Definitions

We define several types of probabilities in this section. The purpose of defining these probabilities is two-fold: **i)** to quantify a program’s *statically inferred* control-flow properties, and **ii)** to be compatible with the parameters of the probabilistic learning model, so that static program analysis results can be integrated with the learning model. The values include the conditional probability of adjacent CFG nodes, the reachability probability from the function entry, and transition probability for a call pair.

Definition 5 *The conditional probability P_{ij}^c of adjacent CFG nodes for a node pair (n_i, n_j) or $(n_i \rightarrow n_j)$ is the probability of occurrence for node n_j , conditioning on that its immediate preceding node n_i has just been executed, i.e., $P[n_j|n_i]$.*

Definition 6 *The reachability probability P_i^r for a CFG node n_i is the likelihood of the function’s control flow reaches node n_i , i.e., the likelihood of n_i being executed within this function.*

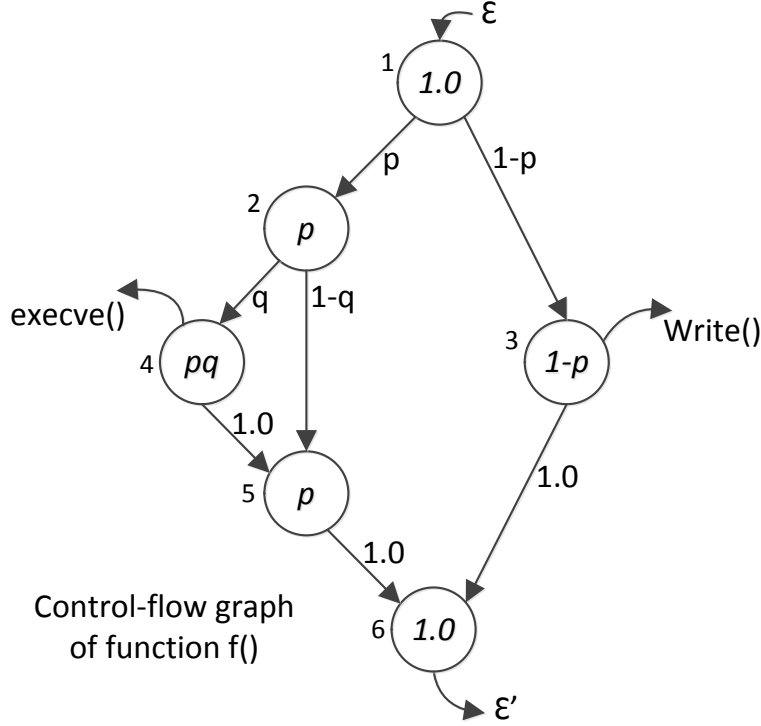


Figure 4.3: Examples of conditional probabilities and reachability probabilities for function $f()$. Conditional probability of a node pair is shown on the edge. Reachability probability of a node is shown in the node. ϵ and ϵ' represent the external call site and return site of $f()$.

Definition 7 The transition probability P_{ij}^{tf} of call pair (c_i, c_j) in function $f()$ is defined as the likelihood of occurrence of the call pair during the execution of the function.

Our method first traverses the control-flow graph of a function to statically approximate the *conditional probability* P_{ij}^c for each pair of adjacent nodes $(n_1 \rightarrow n_2)$. Then, based on conditional probabilities, our algorithm computes the *reachability probability* P_i^r for each node n_i , which represents the likelihood of n_i being executed in the function. Finally, with these reachability probabilities, we compute *transition probabilities* for call pairs.

Attack model: Our detection is aimed at detecting invalid and abnormal control flow of a program, e.g., executing injected code through unsanitized arguments or buffer overflow vulnerabilities, bypassing security checks, exploiting a race condition. These threats may be introduced through human errors (e.g., unauthorized use or operation of the program), software flaws (e.g., buffer overflow vulnerabilities), attacks by remote attackers or malicious insiders (e.g., through drive-by downloads, infecting the system with malicious attachment or devices). A Hidden Markov Model based anomaly detection (e.g., [117]) can also be used to detect symbolic link attack and denial-of-service.

A hand-crafted mimicry attack was first introduced in [113], where the system calls in a malicious action are in an order that is compatible with the detection model. Although our model is not specifically designed to detect general mimicry attacks (which is an open problem), it can catch mimicries that involve the invocation of legitimate-yet-rare calls or paths having low likelihoods of occurrences. The likelihood of occurrence computation in our detection significantly increases the difficulty required for attackers to develop mimicry attack sequences.

We assume that the data collection, model building, and enforcement operations are not tampered with, which is a standard assumption in anomaly detection and can be realized in practice (e.g., by using a virtual machine architecture for monitoring system calls and building the models on a trusted server).

4.2 Probability Forecast of Call Sequences

In this section, we present details of our algorithms for realizing CFG-based control-flow probability forecast. Our concept of probability forecast is new; none of the existing learning-based anomaly detection techniques performs this analysis. Our main advantage is the completeness of the classification model. The model is capable of recognizing new legitimate call sequences not seen during training, thus reducing false negative rate.

Our algorithms statically approximate the likelihoods of occurrence of call sequences in a program, through computing reachability probabilities (Definition 6) and transition probabilities (Definition 7). These values reflect statically inferred program behaviors. Together with the dynamic behaviors learned from program traces, we obtain a comprehensive probabilistic model. (The integration is described in the next section.)

Researchers have proposed probabilistic techniques for solving programming and software problems, such as vulnerability analysis for software testing [102] and fault localization for debugging [7, 8].

For vulnerability analysis, [102] aims to find paths from inputs to sensitive operations. Using the edge probabilities in CFGs, a genetic algorithm creates new generations of inputs which increases the code coverage and depth of penetration into program control flow logic. Baah *et al.* finds dependency among program components using statistical inference techniques for fault localization [8, 7]. The analysis is based on program dependency graph and its probability parameters are learned through execution.

Our static program analysis is based on CFGs and the purpose is to find out all possible call sequences within a program. While the compared work focuses more on particular paths and error executions. Besides the obvious differences in probability semantics, our probability analysis – covering the entire control-flow graphs and call graph of a program – is more comprehensive and rigorous than the ones described in [8, 102]. The latter are limited to

pair-wise conditional probabilities on a control flow graph.

4.2.1 Reachability Probability

To compute *reachability probability* (Definition 6), our algorithm traverses a CFG and estimates the probability to reach a CFG node from the function entry. We assume that the execution is started from the entry of a function with the probability of 1.0. The calculation of reachability probabilities is top down starting from the function entry of CFG. Formally, for node n_k the reachability probability P_k^r is computed as in Equation 4.1, where P_i^r is the reachability probability of one of n_k 's parents and P_{ik}^c is the conditional probability (Definition 5) for node pair (n_i, n_k) .

$$P_k^r = \sum_{\forall n_i \in \text{parent set of } n_k} P_i^r * P_{ik}^c \quad (4.1)$$

Given the CFG of a function in a program, we infer the *conditional probability* of an adjacent node pair based on the configuration of outgoing edges of the parent node. Specifically, P_{ij}^c for node pair (n_i, n_j) is based on the branching factor at the parent node n_i in the control-flow graph. If node n_i has only one child node n_j , then $P[n_j|n_i] = 1$. If n_i has two or more child nodes, P_{ij}^c follows a probability distribution function, e.g., an equal or biased distribution. Advanced branch prediction and path frequency approximation techniques can be utilized, such as branch prediction [10, 18, 120], path frequency [17].

We perform the topological sorting on all nodes and our reachability-probability computation follows the topological order. To compute the probability of a child node, one needs the reachabilities of its parents.

We illustrate the probability values for a simple control-flow graph for a function $f()$ in Figure 4.3. P_5^r for node 5 is computed as $pq * 1 + p * (1 - q) = p$, where pq and p are the reachability probabilities of its two parents, and 1 and $1 - q$ are the conditional probabilities with respect to the two incoming edges of node 5. We assign the reachability probability of each node following the topological order, so that a node inherit its probability from all parents before passing the probability to its children.

Pseudocode of our algorithm for estimating probabilities of reachability is given in Algorithm 1. This pseudocode computes the reachability probability of each node within a control flow graph.

The complexity for computing reachability probabilities for a control-flow graph $G(V, E)$ with nodes V and edges E is $O(|V| + |E|)$. The number of outgoing edges for each node is usually small (e.g., 2 or 3). Thus, the complexity is $O(|V|)$ in practice.

Algorithm 1 Pseudocode for Computing Probabilities of Reachability.

Input: The control flow graph of a function.

Output: The reachability for each node in the function.

```

function PROCESS_REACHABILITY(function)
  /* Topological sort, children ordered after parent */
  topoList = topological_sort(function.nodeList)

  /* Assign and propagate probability */
  n = topoList.begin()
  n.prob = 1.0
  while (!topoList.empty()) do
    n = topoList.pop_back()
    probPerChild = n.prob/n.children.size()
    for (i = 1 → n.children.size()) do
      n.children[i].prob+ = probPerChild
    end for
  end while
end function

```

4.2.2 Call-Transition Matrix of Function

Based on reachability probabilities, one can then compute likelihoods of occurrence for call pairs in a function, i.e., *transition probability* (Definition 7). The call-transition probability is defined for a call pair (c_1, c_2) , where c_1 precedes c_2 .

We define the *call-transition matrix of a function* in Definition 8. Table 4.1 shows the call-transition matrix of the function in Figure 4.3 under the uniform distribution for conditional probabilities (i.e., $p = q = 0.5$).

Definition 8 *Call-transition matrix of a function stores pair-wise call-transition probabilities of the function. The rows and columns of the matrix correspond to calls that appear in the control-flow graph of the function, respectively. A cell (c_i, c_j) stores the likelihood of occurrence for call pair $(c_i \rightarrow c_j)$, i.e., transition probability P_{ij}^t .*

The call-transition matrix of a function or a program needs to satisfy the following laws of probability:

Definition 9 *Rules of call-transition matrix of a function:*

1. The sum of first row of a call-transition matrix of function $f()$ should sum to 1.0, i.e., $\sum_i P_{ei}^{t_f} = 1.0$. This rule means that $f()$ is called with a probability of 1.

2. For each call c_i in a call-transition matrix of function $f()$, the sum of its incoming probabilities should equal the sum of its outgoing probabilities, i.e., $\sum_j P_{ji}^{tf} = \sum_k P_{ik}^{tf}$.

For Table 4.1, the sum of transition probabilities for call pairs (ϵ, ϵ') , (ϵ, write) , $(\epsilon, \text{execve})$ is 1 (following Rule 1). Transition probabilities of call pairs (ϵ, write) and $(\text{write}, \epsilon')$ are equal; $(\epsilon, \text{execve})$ and $(\text{execve}, \epsilon')$ are equal (following Rule 2).

A program may contain multiple functions. Thus, obtaining the call-transition matrix corresponding to the program requires the aggregation of transition probabilities in individual CFG call-transition matrices (described in the next section).

	ϵ'	write	execve
ϵ	0.25	0.5	0.25
write	0.5	0	0
execve	0.25	0	0

Table 4.1: A call-transition matrix of the function in Figure 4.3 under the uniform distribution for conditional probabilities. ϵ represents the external caller of this function. ϵ' represents the external return site.

Many nodes in CFG do not make calls. Thus there may be multiple nodes on the path between two call-making nodes. In those cases, the computation of transition probability needs to properly consider the additional branching factors introduced by those non-call-making nodes. In our prototype the classification is performed on call sequences. If a node does not make calls (e.g., syscall or function call), then the instructions in that node cannot be observed. Thus, the estimation of transition probabilities is performed only for call-making nodes. The non-call-making nodes serve as bridges to connect two call-making nodes.

To compute the transition probability P_{ab}^{tf} of a call pair (c_a, c_b) in $f()$, we identify all the nodes $\{L\}$ such that a node $n_l \in L$ has the following three properties. Let n_k be a node in CFG that makes a call c_a . **i)** node n_l makes a call (e.g., libcall or syscall) c_b , **ii)** there exists a directed path (denoted by $n_k, n_{k+1}, \dots, n_{l-1}, n_l$) from n_k to n_l , and **iii)** no other nodes on the path between n_k and n_l make any calls. Then for each node $n_l \in L$, compute the transition probability $P_{a_k b_l}^{tf}$ of call pair (c_a, c_b) in $f()$ as Equation (4.2).

$$P_{a_k b_l}^{tf} = P_k^r * \prod_{i=k}^{l-1} P_{i(i+1)}^c \quad (4.2)$$

In a context-sensitive model as shown in Equation (4.2), the calling context is recorded when computing a call-transition probability. In a context-insensitive model, the identities of the callers are not recorded. In that case, transition probabilities of all the occurrences of identical call pairs in the function are added up as shown in Equation (4.3). Our STILO prototype

is flow-sensitive and context-insensitive. Thus, the aggregation follows Equation (4.3). Enhancing the sensitivity of calling context is our ongoing work.

$$P_{ab}^{tf} = \sum_{\substack{\forall \text{ node pairs } (n_k, n_l) \\ \text{s.t. } n_k \text{ calls } c_a, n_l \text{ calls } c_b}} P_{a_k b_l}^{tf} \quad (4.3)$$

We process CFG nodes following the reverse topological ordering, which avoids duplicate traversals when searching for call transitions. In addition, node probabilities are computed and cached, which avoids recomputing from scratch. As a result, the worst-case complexity of our algorithm is $O(|E|)$ ¹.

Details and pseudocode of our algorithm for estimating the transition probabilities is given in Algorithm 2. Our algorithm finds the immediate call set (ICS) for each node within a CFG. The immediate call set (ICS) consists of calls that are immediately following the owner node with no other calls in between. Each node n has an ICS n_{ICS} which is a set, and each element within the set is represented as a pair $\langle c_j, P_{nj}^t \rangle$, where c_j is one of the immediate calls of node n and P_{nj}^t is the corresponding transition probability from node n to the immediate call c_j .

We start by looking at the transition between two nodes. For the transition $(n \rightarrow j)$, its transition probability P_{nj}^t is based on the reachability probability P_n^r of the parent node n and the conditional probability P_{nj}^c of the pair, specifically $P_{nj}^t = P_n^r * P_{nj}^c$.

We then find all the transition probabilities from each node to its immediate calls. Within the CFG, we identify every node which makes a call, and each node's ICS is a collection of possible transitions from this node to every immediate call in the CFG. The ICS of node n is computed by summing up the relevant transition probability information of n 's child nodes. If the child node is making a system call or library call, this call is included in the parent node's ICS, otherwise if the child node is not making a system or library call, the child node's ICS is included into the parent node's ICS with certain probabilities.

This step finds the immediate call set for each node within a CFG. We get the ICS (immediate call set) for each node in reverse topological order, so that whenever we construct the ICS for a current node, its children's ICSs have already been processed. The current node's ICS can be constructed using the ICS information from all its children nodes.

$$n_{ICS} = \left(\sum_i \frac{P_n^r * P_{ni}^c}{P_i^r} * n[i]_{ICS} + \sum_j \langle n[j].call, P_n^r * P_{nj}^c \rangle \right) \quad (4.4)$$

Formally, Equation 4.4 computes the transition probability for n . Node i is a child node of node n that does not make a call. Node j is a child node of n that makes a call. P_n^r and

¹The complexity is close to $O(|V|)$ in practice, as most nodes have less than 3 outgoing edges.

P_j^r are reachability probabilities of n and j , respectively. P_{ni}^c and P_{nj}^c are the conditional probabilities for node pairs $(n \rightarrow i)$ and $(n \rightarrow j)$, respectively.

Following the reverse topological order, each node is processed only once, and the ICS for each node is constructed using the information from its children nodes, which also leads to a run-time complexity of $O(|V|)$. For our running example in Figure 4.3, the ICSs of all nodes are listed in Table 4.2.

Node	Immediate Call Set
6	$\{< \epsilon, 1.0 >\}$
5	$\{< \epsilon, 0.5 >\}$
4	$\{< \epsilon, 0.25 >\}$
3	$\{< \epsilon, 0.5 >\}$
2	$\{< \epsilon, 0.25 >, < execve, 0.25 >\}$
1	$\{< \epsilon, 0.25 >, < execve, 0.25 >, < write, 0.5 >\}$
ϵ	$\{< \epsilon, 0.25 >, < execve, 0.25 >, < write, 0.5 >\}$

Table 4.2: Examples of immediate call sets.

4.2.3 Aggregation of Call Transitions

We need a way to obtain a more compact representation out of the collection of call transition matrices we have obtained from the last step. The purpose of AGGREGATION OF CALL TRANSITIONS operation is to aggregate multiple call-transition matrices, each corresponding to a function, into one (larger) complete call-transition matrix representing the entire program. This complete matrix is then used to statically initialize the Markov-based learning model. The input is a complete set of call transition matrices associated with the program. The output is a complete call transition matrix of the program that incorporates all the possible call transitions that may occur in the program.

Specifically, the operation takes as inputs **i)** the call graph of the program and **ii)** call-transition matrix for each function. The call graph is need for the calling relations among functions. The operation has two tasks:

- *To extend and connect the individual control flows:* This task is realized by inlining the call-transition matrices of callee functions into those of caller functions, and augmenting the rows and columns of the call-transition matrix. The call relations are obtained from the call graph of the program.
- *To update transition probabilities:* This task involves two types of computation: *multiplication* to adjust the reachability, and *addition* to aggregate probabilities of identical call pairs across the program.

Algorithm 2 Immediate call set construction for each node within CFG.

Input: The control flow graph of a function after reachability estimation.

Output: The ICS for each node in the function.

```

function PROCESS_ICS(function)
  /* Reverse Topological sort, children ordered before parent */
  rTopoList = reverse_topological_sort(function.nodeList)

  /* Construct ICS for each node */
  while (!rTopoList.empty()) do
    n = rTopoList.pop_back()
    if (n.children.empty()) then
      nICS = {<  $\epsilon$ ,  $P_n^r$  >}
      continue
    end if
    for ( $i = 1 \rightarrow n.children.size()$ ) do
       $P_{ni}^c = 1.0/n.children.size()$ 
      if (n.children[i].makeCall) then
         $n_{ICS} += \frac{P_n^r * P_{ni}^c}{P_i^r} * P_i^r * n.children[i].call$ 
      else
         $n_{ICS} += \frac{P_n^r * P_{ni}^c}{P_i^r} * ICS_i$ 
      end if
    end for
  end while
end function

```

Our approach is to inline the call transition matrices of callee functions into the matrices of caller functions, so that:

- i) The final call transition matrix captures the execution pattern of the whole program instead of single functions. This call transition matrix is later used in HMM initialization for the whole program.
- ii) The final transition matrix consists of only system calls or library calls. All other function calls are reduced through the merging process.

The aggregated call-transition matrix should also satisfy the rules of probability in Definition 9.

We distinguish three cases of call pairs during the probability aggregation (as illustrated in Figure 4.4).

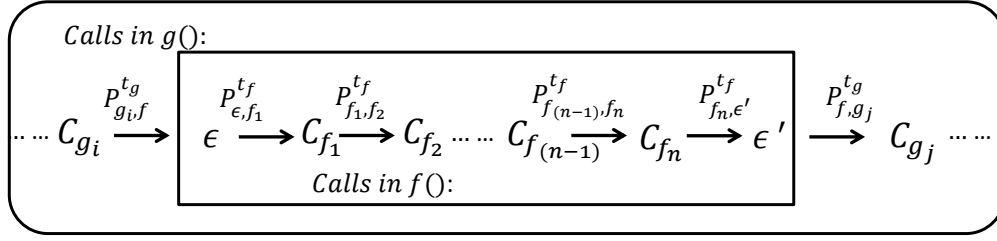


Figure 4.4: Illustration of call sequences in a caller function $g()$ and a callee function $f()$. Indices are from topological sort. The AGGREGATION operation replaces and expands entries with $f()$ in $g()$'s call-transition matrix with calls in $f()$.

Suppose that function $f()$ is called within function $g()$. **i)** Call pairs $(c_{g_i}, f())$ and $(f(), c_{g_j})$, where c_{g_i} and c_{g_j} are calls in function $g()$ that immediately precede and immediately follow the call to $f()$, respectively. **ii)** $(\epsilon, f())$, i.e., there is no call made in $g()$ that immediately precedes the call to $f()$. **iii)** $(f(), \epsilon')$, i.e., there is no call made in $g()$ that immediately follows the call to $f()$.

Details of computation are described in pseudocode in Algorithm 3. This matrix output by AGGREGATION quantitatively represents the pair-wise control flow of the program obtained through the static program analysis. The worst-case complexity of AGGREGATE is linear in the total number of adjacent call pairs in the program and the number of edges in the call graph of the program.

An example is given in the Section 4.2.4. We give the details about the algorithms here.

Details of Aggregation Algorithm:

- For each call c_{f_k} appearing in the first row of $f()$'s call-transition matrix (i.e., pairs $(*, f())$ with $f()$ being the child node), there are two cases. **i)** If pair (c_{g_i}, c_{f_k}) does not exist in $g()$'s transition matrix, then add a column for c_{f_k} in $g()$'s call-transition matrix, and let the new transition probability (c_{g_i}, c_{f_k}) be $P_{g_i,f_k}^{t_g} = P_{g_i,f}^{t_g} * P_{\epsilon,f_k}^{t_f}$, where $P_{\epsilon,f_k}^{t_f}$ is the transition probability in $f()$ associated with call pair (ϵ, c_{f_k}) . **ii)** Otherwise, compute the new transition probability of call pair (c_{g_i}, c_{f_k}) as $P_{g_i,f_k}^{t_g} + P_{g_i,f}^{t_g} * P_{\epsilon,f_k}^{t_f}$, where $P_{g_i,f_k}^{t_g}$ is transition probability in $g()$ for pair (c_{g_i}, c_{f_k}) before aggregation.
- For each call c_{f_l} appearing in the first column of $f()$'s call-transition matrix (i.e., call pairs $(f(), *)$ with $f()$ being the parent node), we distinguish two cases. **i)** If pair (c_{f_l}, c_{g_j}) does not exist in $g()$'s transition matrix, then add a row for c_{f_l} in $g()$'s call-transition matrix and let the new transition probability of pair (c_{f_l}, c_{g_j}) be $P_{f_l,g_j}^{t_g} * P_{f_l,\epsilon'}^{t_f}$, where $P_{f_l,\epsilon'}^{t_f}$ is the transition probability in $f()$ associated with pair (c_{f_l}, ϵ') . **ii)** Otherwise, update the transition probability in $g()$ for pair (c_{f_l}, c_{g_j}) as $P_{f_l,g_j}^{t_g} + P_{f_l,\epsilon'}^{t_f} * P_{\epsilon',g_j}^{t_g}$.

$P_{f_l, \epsilon'}^{t_f}$, where $P_{f_l, g_j}^{t_g}$ is the probability in $g()$'s matrix before aggregation.

- Each of the other call pairs (c_{f_k}, c_{f_l}) in $f()$ with transition probability $P_{f_k, f_l}^{t_f}$ is aggregated into $g()$'s transition matrix: **i)** If the call pair (c_{f_k}, c_{f_l}) does not exist in $g()$'s transition matrix, add column and row for c_{f_k} and c_{f_l} and compute the new transition probability of (c_{f_k}, c_{f_l}) as $P_{f_k, f_l}^{t_g} = (\sum_i P_{g_i, f}^{t_g}) * P_{f_k, f_l}^{t_f}$. **ii)** Otherwise, compute the new transition probability for (c_{f_k}, c_{f_l}) as $P_{f_k, f_l}^{t_g} + (\sum_i P_{g_i, f}^{t_g}) * P_{f_k, f_l}^{t_f}$, where $P_{f_k, f_l}^{t_g}$ is the probability in $g()$'s matrix before the aggregation.
- If function $f()$ does not make any calls, then compute the new transition probability for pair (c_{g_i}, c_{g_j}) in $g()$ after the aggregation as: $P_{g_i, g_j}^{t_g} + P_{g_i, f}^{t_g} * P_{\epsilon, \epsilon'}^{t_f} * P_{f, g_j}^{t_g}$, where $P_{g_i, g_j}^{t_g}$ is the transition probability in $g()$ for pair (c_{g_i}, c_{g_j}) before the aggregation.
- Remove the row and column in the call-transition matrix of $g()$ that corresponds to $f()$.

Order of aggregation Given the individual call-transition matrices of functions in the program, the order of aggregation follows a reverse topological ordering in the call graph. First, one obtains the topological order $(f_1(), f_2(), \dots, f_n())$ of all internal functions of a program based on their call relations specified in the call graph; then performs AGGREGATE operation by aggregating $f_i()$'s matrix into $f_{i-1}()$ for $i = n, \dots, 2$.

One can prove that the matrix produced by our algorithm satisfies the probability rules in Definition 9. The proof is below:

There are 2 properties associated with the call-transition matrices and are preserved throughout the aggregate operations.

- 1. The sum of first row/column of a call-transition matrix of function $f()$ should add up to 1.0, i.e., $\sum_k P_{j\epsilon k}^{t_f} = 1.0$ and $\sum_j P_{j\epsilon k}^{t_f} = 1.0$.
- 2. For each call f_i in a call-transition matrix of function $f()$, the sum of its incoming probabilities should be equal to the sum of its outgoing probabilities, i.e., $\sum_j P_{ji}^{t_f} = \sum_k P_{ik}^{t_f}$.

These 2 properties that are consistent before and after our aggregate operations. The proof is given here. Consider the step where function f_n is called within function f_m , so the call-transition matrix of f_n is merged into the call-transition matrix of function f_m during the aggregate operation.

For property 1, take the first row of function f_m for example. For the case of column, the property can be proved in the same way. After one aggregate operation between caller

function f_m and callee function f_n , the new sum of the the probabilities of the first row is:

$$\begin{aligned}
 P_{1st_row_after}^{f_m} &= \sum_{k \neq k_{f_n}} P_{\epsilon k}^{t_{f_m}} \\
 &+ P_{\epsilon k_{f_n}}^{t_{f_m}} * \left(\sum_{k \neq k_{\epsilon'}} P_{\epsilon k}^{t_{f_n}} \right) \\
 &+ P_{\epsilon k_{f_n}}^{t_{f_m}} * P_{\epsilon \epsilon'}^{t_{f_n}} * \left(\sum_k \frac{P_{j_{f_n} k}^{t_{f_m}}}{\sum_l P_{j_{f_n} l}^{t_{f_m}}} \right)
 \end{aligned} \tag{4.5}$$

In equation 4.5, part 1 represents the transition probabilities that are not related to callee function f_n , and part 2 represents the transition probabilities that are added based on the first row of f_n 's transition matrix due to aggregate. Part 3 includes the added transition probabilities when callee function f_n makes no call.

Equation 4.5 further goes as:

$$\begin{aligned}
 P_{1st_row_after}^{f_m} &= \sum_{k \neq k_{f_n}} P_{\epsilon k}^{t_{f_m}} + P_{\epsilon k_{f_n}}^{t_{f_m}} * \left(\sum_{k \neq k_{\epsilon'}} P_{\epsilon k}^{t_{f_n}} \right) \\
 &+ P_{\epsilon k_{f_n}}^{t_{f_m}} * P_{\epsilon \epsilon'}^{t_{f_n}} * 1.0 \\
 &= \sum_{k \neq k_{f_n}} P_{\epsilon k}^{t_{f_m}} + P_{\epsilon k_{f_n}}^{t_{f_m}} * \left(\sum_k P_{\epsilon k}^{t_{f_n}} \right) \\
 &= \sum_{k \neq k_{f_n}} P_{\epsilon k}^{t_{f_m}} + P_{\epsilon k_{f_n}}^{t_{f_m}} * 1.0 \\
 &= \sum_k P_{\epsilon k}^{t_{f_m}} = P_{1st_row_before}^{f_m} \\
 &= 1.0
 \end{aligned} \tag{4.6}$$

For property 2, consider a call f_x in function f_m , $f_x \neq f_n$. After aggregate operation, the outgoing probability for f_x is:

$$\begin{aligned}
 P_{out_after}^{f_x} &= \sum_{k \neq k_{f_n}} P_{j_{f_x} k}^{t_{f_m}} \\
 &+ P_{j_{f_x} k_{f_n}}^{t_{f_m}} * \left(\sum_{k \neq k_{\epsilon'}} P_{j_{\epsilon} k}^{t_{f_n}} \right) \\
 &+ P_{j_{f_x} k_{f_n}}^{t_{f_m}} * P_{j_{\epsilon} k_{\epsilon'}}^{t_{f_n}} * \left(\sum_k \frac{P_{j_{f_n} k}^{t_{f_m}}}{\sum_l P_{j_{f_n} l}^{t_{f_m}}} \right)
 \end{aligned} \tag{4.7}$$

Similarly, In equation 4.7, for the right side, part 1 represents the transition probabilities that are not related to callee function f_n , and part 2 represents the transition probabilities

that are added based on the first row of f_n 's transition matrix due to aggregate. Part 3 includes the added transition probabilities when callee function f_n makes no call.

Equation 4.7 can be reduced as:

$$\begin{aligned}
P_{out_after}^{f_x} &= \sum_{k \neq k_{f_n}} P_{j_{f_x}k}^{t_{f_m}} + P_{j_{f_x}k_{f_n}}^{t_{f_m}} * \left(\sum_{k \neq k_{\epsilon'}} P_{j_{\epsilon}k}^{t_{f_n}} \right) \\
&\quad + P_{j_{f_x}k_{f_n}}^{t_{f_m}} * P_{j_{\epsilon}k_{\epsilon'}}^{t_{f_n}} * 1.0 \\
&= \sum_{k \neq k_{f_n}} P_{j_{f_x}k}^{t_{f_m}} + P_{j_{f_x}k_{f_n}}^{t_{f_m}} * \left(\sum_k P_{j_{\epsilon}k}^{t_{f_n}} \right) \\
&= \sum_{k \neq k_{f_n}} P_{j_{f_x}k}^{t_{f_m}} + P_{j_{f_x}k_{f_n}}^{t_{f_m}} * 1.0 \\
&= \sum_k P_{j_{f_x}k}^{t_{f_m}} = P_{out_before}^{f_x}
\end{aligned} \tag{4.8}$$

Similarly, for incoming probabilities we also have $P_{in_after}^{f_x} = P_{in_before}^{f_x}$. So after aggregate operation, we still have:

$$P_{in_after}^{f_x} = P_{in_before}^{f_x} = P_{out_before}^{f_x} = P_{out_after}^{f_x} \tag{4.9}$$

Summary In STILO (**ST**atically **I**nitialized mark**O**v), the analysis described in this section takes as inputs control flows that can be statically inferred, and transforms them into a rigorous probability representation. Program behaviors that cannot be covered by static program analysis (e.g., function pointer) will be learned from program traces by our STILO HMM model. We describe how STILO HMM utilizes the obtained probability values next.

4.2.4 An Aggregation Example

Suppose function $g()$ calls function $f()$ as in Figure 4.5. $g()$'s call-transition matrix may be:

$$\begin{pmatrix}
& \epsilon' & \text{getuid} & \text{read} & f() & \text{execve} \\
\epsilon & 0 & 0.5 & 0.5 & 0 & 0 \\
\text{getuid} & 0 & 0 & 0 & 0 & 0.5 \\
\text{read} & 0 & 0 & 0 & 0.5 & 0 \\
f() & 0 & 0 & 0 & 0 & 0.5 \\
\text{execve} & 1 & 0 & 0 & 0 & 0
\end{pmatrix}$$

The new call-transition matrix for function $g()$ after AGGREGATION should be:

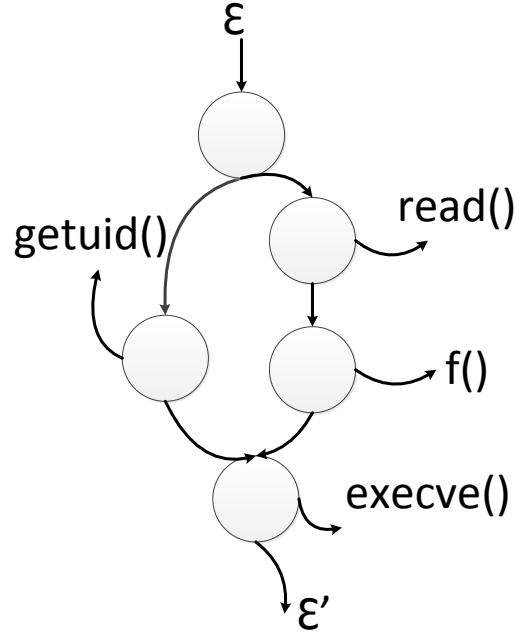


Figure 4.5: CFG for caller function $g()$. Function $f()$ is called within function $g()$.

$$\begin{pmatrix}
 & \epsilon' & \text{getuid} & \text{read} & \text{write} & \text{execve} \\
 \epsilon & 0 & 0.5 & 0.5 & 0 & 0 \\
 \text{getuid} & 0 & 0 & 0 & 0 & 0.5 \\
 \text{read} & 0 & 0 & 0 & 0.25 & 0.25 \\
 \text{write} & 0 & 0 & 0 & 0 & 0.25 \\
 \text{execve} & 1 & 0 & 0 & 0 & 0.125
 \end{pmatrix}$$

During the aggregation, we compute and re-assign the transition probabilities for caller function $g()$. As an example, we explain how the new probabilities are obtained for the *read* row in the aggregated new call transition matrix.

- Before the aggregation, there's no transition of *read()* $\rightarrow$ *write()* in $g()$. However, the transition probability of *read()* $\rightarrow$ $f()$ in caller function $g()$ is 0.5. Inside the the callee function $f()$, the transition probability of ϵ $\rightarrow$ *write()* is 0.5. As a result, the new transition probability of *read()* $\rightarrow$ *write()* for caller function $g()$ after aggregation is $0.5 * 0.5 = 0.25$.
- Before the aggregation, the transition probability of *read()* $\rightarrow$ *execve()* in caller function $g()$ is 0. However, the transition probability of *read()* $\rightarrow$ $f()$ in caller function $g()$ is 0.5. Inside the the callee function $f()$, the transition probability of ϵ $\rightarrow$ *execve()* is 0.25. This adds $0.5 * 0.25$ to the transition probability of *read()* $\rightarrow$ *execve()* in the

aggregated matrix. Also $f()$ has the probability 0.25 to make no call. Since in $g()$, $f()$ only returns back to $execve()$. This adds another $0.5 * 0.25$ to the transition $read() \rightarrow execve()$. As a result, the final transition probability for $read() \rightarrow execve()$ in the aggregated call transition matrix is the sum of two parts, thus 0.25.

The aggregation operations cover the transition from caller function to callee function, then the return from callee function back to caller function, and update the call transition probabilities inside the caller function’s new call transition matrix.

4.2.5 HMM Initialization

Our hidden Markov model is initialized with the call-transition probabilities and call information obtained from the static program analysis. The initialized HMM still needs to be trained with program traces to learn and cover dynamic control-flow patterns. Our final HMM represents **both static and dynamic prediction** of the program’s behaviors.

Hidden state We give semantic meanings to the initial hidden states. We let them represent the logical reasons (or program phases) governing the actions of a program. In our prototype, we associate each hidden state with a distinct system call or library call in the *aggregated call-transition matrix*. Therefore, there is a one-to-one correlation between hidden states and calls in the program. In our STILO prototype, the number N of hidden states is the total number of distinct calls in the program code. This design choice enables us to conveniently incorporate statically obtained information into HMM. In regular HMMs (e.g., [117, 124]), N is the approximated number of distinct calls in program traces (which is usually smaller²).

Observation symbol The observation symbols M need to be associated with observable program behaviors. We define the *observation symbols* as system calls or library calls.

Emission probability Because of the semantics of our hidden states, it is straightforward to initialize the emission probabilities. For each hidden state i , we assign a high emission probability (e.g., 0.5) for the call that i corresponds to, and assign random low probabilities to the rest of the observation symbols.

State-transition probability Our HMM’s state-transition probabilities $\{A\}$ are initialized with the transition probabilities $\{P_{ij}^t\}$ of call pairs in the program’s aggregated call-transition matrix.

Initial probability distribution In STILO, because of our one-to-one correlation between hidden states and calls, the distribution π of hidden states is approximated based on the program’s call-transition matrix. Specifically π_i is initialized with the frequencies of call occurrences ($\sum_j P_{ij}^t$) and normalized.

²Our experiments show a larger N sometimes improves the classification performance, but it is not guaranteed.

4.3 Evaluation of Statically Initialized Model

Our prototype for the static program analysis is implemented in C/C++ using the **Dyninst** library (<http://www.dyninst.org/>). Then the major task is processing the output control flow graphs and extracting call sequence information for constructing an initial base Hidden Markov Model. The HMM training and evaluation code is written in Java using the **Jahmm** library (<https://code.google.com/p/jahmm/>), which gives an Java implementation of the Viterbi, Forward-Backward, Baum-Welch and K-Means algorithms and so on. We refer to our prototype *STILO*, short for **ST**atically **Initia**Lized mark**OV**. We use the system tools *strace* and *ltrace* to intercept system calls and library calls of running application processes. There are over 200 distinct system calls and over 1,000 distinct library calls available in a normal operating system. System calls may be wrapped up in different names when they are exported to user space. We obtain the wrapper functions for system calls from the **glibc** source code. The library calls of interest are the **glibc** library calls, which are a collection of C standard libraries. We identify these calls by looking up their names in the control flow graphs and call graph.

4.3.1 Experimental Setup

Program	size	Lines of code (c)	Lines of code (header)	Lines of code (total)
bash	2.2MB	61020	3972	64992
flex	192.3 kB	9922	387	10309
grep	135.1 kB	9002	1066	10068
gzip	81.1 kB	5096	586	5682
sed	151.2 kB	12884	1538	14422
vim	779.7 kB	108839	4462	113301
proftpd	665.2 kB	107709	4119	111828
nginx	4.9M	83772	6314	90086
Average	1139.1 kB	49780.5	2805.5	47228.86

Table 4.3: Statistics of the size of programs in experiment.

The programs and test cases used in our experiments include utility applications (**flex**, **grep**, **gzip**, **sed**, **bash**, **vim**) from the Software-artifact Infrastructure Repository (SIR) [94], an FTP server **proftpd** and HTTP server **nginx**. The size information is in Table 4.3. Coverages of the test cases are summarized in Table 4.4. With the test cases available in SIR, all the programs get around 50% to 80% coverage for both branches and statements.

Statistics of calls observed during execution and calls in the program code are shown in Tables 5.4 and 5.5 in Section 5.3 of Chapter 5. From left to right, the tables include *i*) the number of unique function calls during program analysis, *ii*) the number of unique

Program	# of test cases	Branch coverage	Line coverage
flex	525	81.3%	76.0%
grep	809	58.7%	63.3%
gzip	214	68.5%	66.9%
sed	370	72.3%	65.6%
bash	1061	66.3%	59.4%
vim	976	55.0%	51.9%
Average	659	67.0%	63.9%

Table 4.4: Statistics of programs and test cases used in experiment. The collected traces are broken into 15-grams (segments) for classification.

library/system calls during program analysis and *iii*) the number of unique library/system calls in the traces that are obtained by running the programs on all the test cases.

We compare the classification performance of STILO with the widely accepted HMM-based classification (e.g., [117, 124]). We refer to that model as the regular HMM model. For the regular HMM, the set of observation symbols consists of distinct calls from execution traces. The number of hidden states is the size of the call set (i.e., the total number of distinct calls in the traces). The regular model randomly chooses the initial HMM parameters, including the initial transition probabilities, initial emission probabilities, and the initial distribution of hidden states.

For our model in the experiment, the number of hidden states is the distinct calls in the matrix. For the [117], it says a good choice of number is roughly equal to the calls used by the program, and in experiment they use 20, 35, 40, or 60 for different programs. In [124] the maximum number of states tested is approximately equal to the total number of different system call categories in the corresponding program (30 for ps, 48 for login, 50 for named, and 60 for sendmail). Both of these papers are choosing the number of states based on the distinct calls in the trace, which is the setting for regular HMM models in our experiment.

Our experiments answer the following questions.

1. How much improvement in classification accuracy does STILO HMM provide compared to the regular HMM? (In Section 4.3.4)
2. What are the reasons for STILO HMM improvement? (In Section 4.3.5)
3. Can STILO detect real-world attack traces, in particular the advanced attacks that introduce subtle control-flow anomalies? (In Section 4.3.6)
4. Which type of traces gives more accurate classification, library calls or system calls, and why? (In Section 4.3.4)

5. What is the impact of the choice of call segment length on classification accuracy? (In Section 4.3.3)

4.3.2 Dynamic Model Training and Testing

Each HMM is specified by above mentioned five sets of parameters, which we use λ for convenient notation. We assign the initial λ with data from static program analysis. Certain degree of inaccuracy exists in the model derived from static analysis since assumptions have been made in terms of probabilities and application's execution paths. The purpose of dynamic modeling is to improve the HMM by applying training call sequences observed from executing the target application.

We show one example in Figure 4.6. Static analysis identifies four possible execution paths from this partial control flow graph, including $\langle read, open, execve \rangle$, $\langle read, open, write \rangle$, $\langle setuid, open, execve \rangle$ and $\langle setuid, open, write \rangle$. However, during actual execution, the only possible path may be $\langle read, open, execve \rangle$ and $\langle setuid, open, write \rangle$.

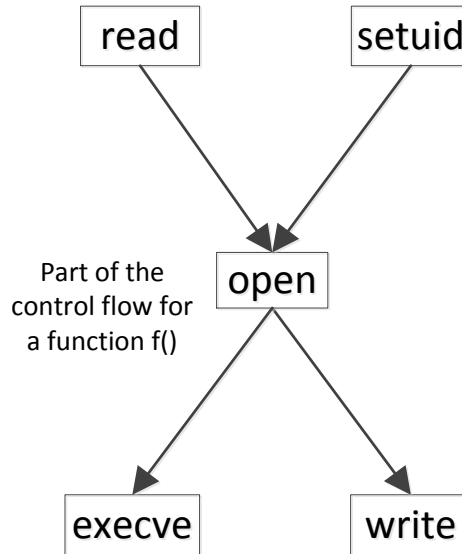


Figure 4.6: Part of the control flow for a function.

This dynamic training process adjusts the model parameters λ to maximize $P(Q|\lambda)$, here Q is the set of training call sequences. Our test included use two levels of observations of applications' behavior, one at system call level and one at library call level. We compare the resulting models of these two approaches in terms of their speed of convergence as well as model accuracy.

A summary of all tested applications is in Table 4.5. For command line applications such as `gzip` and `grep`, we test the executable by running the test cases in SIR. For a server

application (`proftpd`), we test it by connecting to the running server from a client, navigating the server directories, creating new directories, and downloading/uploading/deleting files and folders. For `nginx`, our test cases include both static webpages and dynamic php webpages which interact with an SQL database we set up. Our test cases cover different media types including text, images, scripts and video files with Flash and Mp4 formats. Normal `http` and encrypted `https` accesses are also tested.

Application	Description
gzip	application for file compression and decompression
grep	utility for searching plain-text data sets for lines matching a regular expression
sed	a stream editor to perform basic text transformations on an input stream
flex	a tool for generating scanners for recognizing lexical patterns in text.
vim	a popular open source text editor
bash	a Unix shell program
proftpd	one of the most popular FTP servers
nginx	one of the most popular HTTP servers

Table 4.5: Description of applications used in experiment.

Observed traces are not directly used for model training. We use a window of fixed size to slide along the traces and identify all the trace segments. The window moves one step ahead each time and in this way we generate all the trace segments to be used for both training and validation. Duplicate trace segments are removed, in order to give fair training opportunities for different normal trace segments, otherwise the trained models tend to be over adapted to only the duplicate traces. We also test the influence of different segment lengths for model accuracy.

We divide the collected data set into the following subsets:

- Cross Validation Set (CVS): trace segments for cross validation. We perform 10-fold cross validation on the prepared data set. Each time, 90% of the CVS data is used for model training and the 10% left is used to validate the model after training. So the CVS is further divided into training data set and testing data set for each cross validation.
- Training Termination Set (TTS): trace segments for determining the termination of training iterations. The originally obtained data set is divided into CVS and TTS. To avoid overfit of the model on the training data set, after each iteration of training, we compute the probabilities of the current model on TTS. The training process is stopped when the increase of average probability on TTS is less than a threshold($1.0E-4$) between the current and previous models.

Standard HMM procedures are followed. All comparable HMM models are subject to the same convergence criteria during training. For both STILO and regular HMM models, we

perform *10-fold cross validation* on normal segments. Convergence status is determined by the test results of trace segments in the training termination set. Given a threshold T for a program, false negative (FN) and false positive (FP) rates in HMM are defined in Equations (4.10) and (4.11), where $\{S_A\}$ and $\{S_N\}$ denote the set of abnormal segments and the set of normal segments of the program, respectively, and P_{S_A} and P_{S_N} represent the probability of an abnormal segment and a normal segment, respectively.

$$FN = \frac{|\{S_A : P_{S_A} > T\}|}{|\{S_A\}|} \quad (4.10)$$

$$FP = \frac{|\{S_N : P_{S_N} < T\}|}{|\{S_N\}|} \quad (4.11)$$

Training and classification are on n -grams of program traces, where $n=15$ in our experiments (i.e., all segments consist of 15 calls). Duplicate segments are removed in our training datasets in order to avoid bias.³

We found that classification with segments of length 15 produces more precise results than shorter segments (e.g., length 9) in Section 4.3.3. Therefore, we use 15 as the segment length for our experiments.

Here is a summary of data used for experiment:

- *Normal* segments are obtained by running the target executable and recording the library call or system call segments as the result of the execution. A total of 130,940,213 such segments from more than 4,000 test cases of eight programs are evaluated. These segments are from the Cross Validation Set (CVS).

A HMM classification model needs to give high probabilities to these normal sequences. The training of hidden Markov models requires normal sequences, not abnormal sequences. We test the trained models with two types of *abnormal call sequences*. These sequences should give 0 or low probabilities.

- *Abnormal-A* segments (or attack segments) are obtained by reproducing several real-world attack exploits and payloads. A total of 30,079 such segments are evaluated⁴.
- *Abnormal-S* segments (or synthetic abnormal segments) are generated by replacing the last third of a normal call segment with randomly ordered calls from the legitimate call set. The call set consists of the distinct calls in a program's traces. A total of 160,000 *Abnormal-S* segments are evaluated. Our use of *Abnormal-S* segments enables a rigorous and comprehensive accuracy assessment.

³Experiments were conducted on a Linux machine with Intel Core i7-3770 CPU (@3.40GHz) and 16G memory.

⁴Not all of *Abnormal-A* segments contain exploits.

We assume that all the synthesized segments are abnormal. Some synthetic segments have subtle differences from normal ones and are difficult to detect. Exhausting all possible executions of a program is hard, so in our testing, the normal sequences used are a subset of the theoretical normal sequences set. Also the abnormal sequences used are synthesized by replacing part of the normal sequences with legitimate calls. It's possible that some synthetic sequences may actually be normal sequences but are rarely executed, and thus have not been observed by us. In our experiments, we do not try to identify those cases, and we use only observed normal sequences and synthesized abnormal sequences for model testing.

4.3.3 Impact of Segment Length

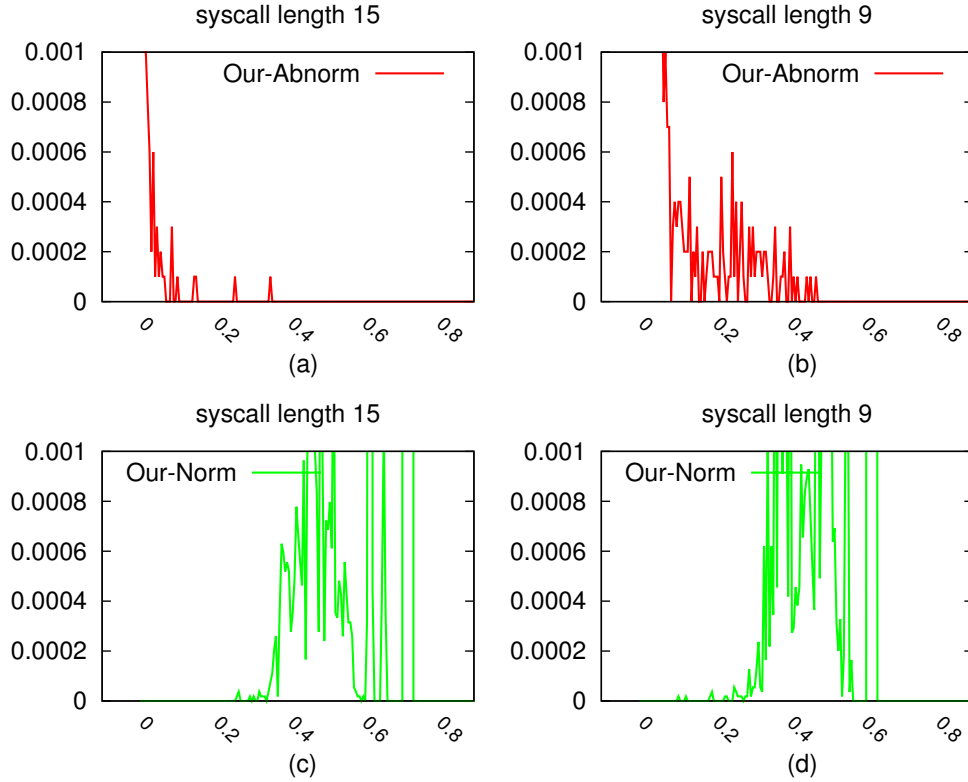


Figure 4.7: HMM classification accuracy in terms of the probability distribution of normal and abnormal syscall segments of lengths 15 in (a)(c) and 9 in (b)(d) for gzip in our converged models. (c) and (d) give a zoomed-in look of (a) and (b) respectively. X-axis represents probability ranging from 0 to 1. Y-axis represents the number of segments. Normal segments should have high probabilities, whereas abnormal segments should be low probability.

Different segment lengths are used for training and testing. Figure 4.7 shows the comparison of model accuracy at convergence for gzip with system call segments of lengths 15 in (a)

and 9 in (b). The accuracy is shown in terms of the separation of probability distribution of normal and abnormal segments, high for normal and low for abnormal ones. A probability threshold is needed for making classification decisions in production systems. Normal `gzip` system call traces and abnormal traces of type Abnormal-S are evaluated.

Results show that the model with 15-call segments (in Figure 4.7 (a) and (c)) gives much better separations between normal and abnormal segments than the one with 9-call segments ((b) and (d)). This finding is consistent with that in [109]. In general, longer call segments can better preserve the normal patterns of call sequences. This property holds for both syscalls and libcalls, as well as for both types of abnormal traces. We use 15 as the segment length for the rest of our experiments.

As an example, for `gzip`, with 0.263 as the threshold at length 15, all the normal syscall trace segments are correctly recognized, i.e., the false positive rate (FP) is 0. With the same threshold, abnormal traces generate a false negative rate of 0.01% – 1 out of 10000 abnormal segments tested has the probability above the threshold.

4.3.4 Classification Accuracy

For each program, we compare STILO and regular HMM’s abilities to recognize new normal segments that do not appear in the training set through 10-fold cross validation with *Normal* segments. We also compare their abilities to recognize *Abnormal-S* segments.

HMM computes the probability of occurrence for each segment. The classification decision is made with respect to a probability threshold T . Different choices of T yield different false positive (FP) and false negative (FN) rates. We show the results of `bash` in Figure 4.8 as an example.

Figure 4.9 shows the averaged false positive and false negative rates of STILO and the regular HMM for syscalls and libcalls. The average is computed across six programs `flex`, `grep`, `gzip`, `sed`, `bash`, and `vim`. The FP rates are in the X-axis. The FN rates in Y-axis are in base-10 log scale. Standard errors are shown as the whisker lines.

STILO consistently demonstrates lower false negative rates than the regular model, when compared with respect to the same false positive rate. This trend is observed for both library calls and system calls. This evidence shows a significantly improved ability in distinguishing normal and abnormal segments when using STILO HMM. **STILO HMM provides 11- to 28-fold improvement in classification accuracy on average compared to the regular HMM.** Figure 4.10 shows the detailed comparisons of false positive and false negative rates between our initialized models and regular models, on both system call and library call segments, and for each program and call type.

From Figure 4.10 it is observed that, false negative number drops when false positive number increases, which is shown across all applications, system calls and library calls. The shape

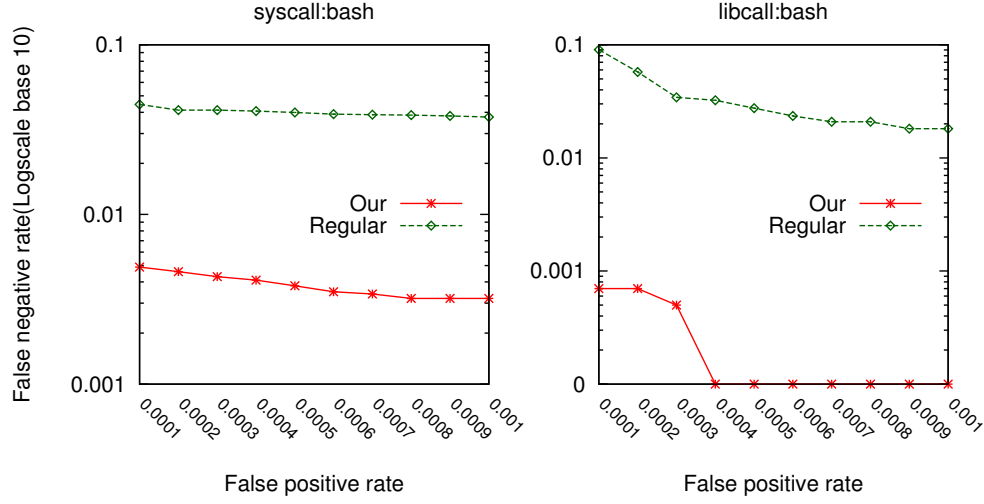


Figure 4.8: Comparison of STILO and regular HMM's false negative rates (in Y-axis, base-10 log scale) for program bash on system calls and library calls under the same false positive rates (in X-axis).

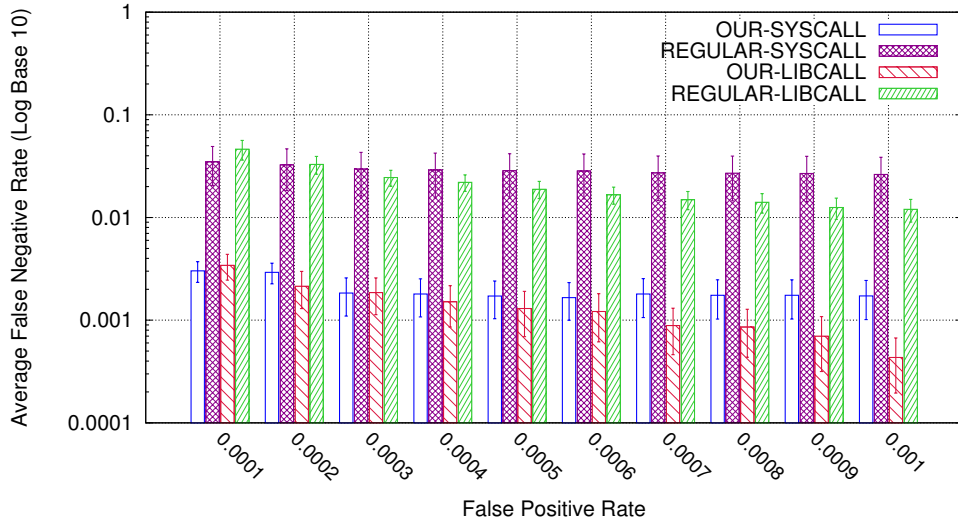


Figure 4.9: Comparison of the averaged false negative rates (in Y-axis) across the six programs from SIR on system call or library calls by our model and the regular HMM with respect to the false positive rates (in X-axis). Standard errors are shown.

of the lines, however, varies depends on several factors:

- First, it depends on which level of monitoring is used. Library calls are made in user space which are more function oriented, while system calls are made to the kernel, which are more atomic and general. And there are more distinct library calls than

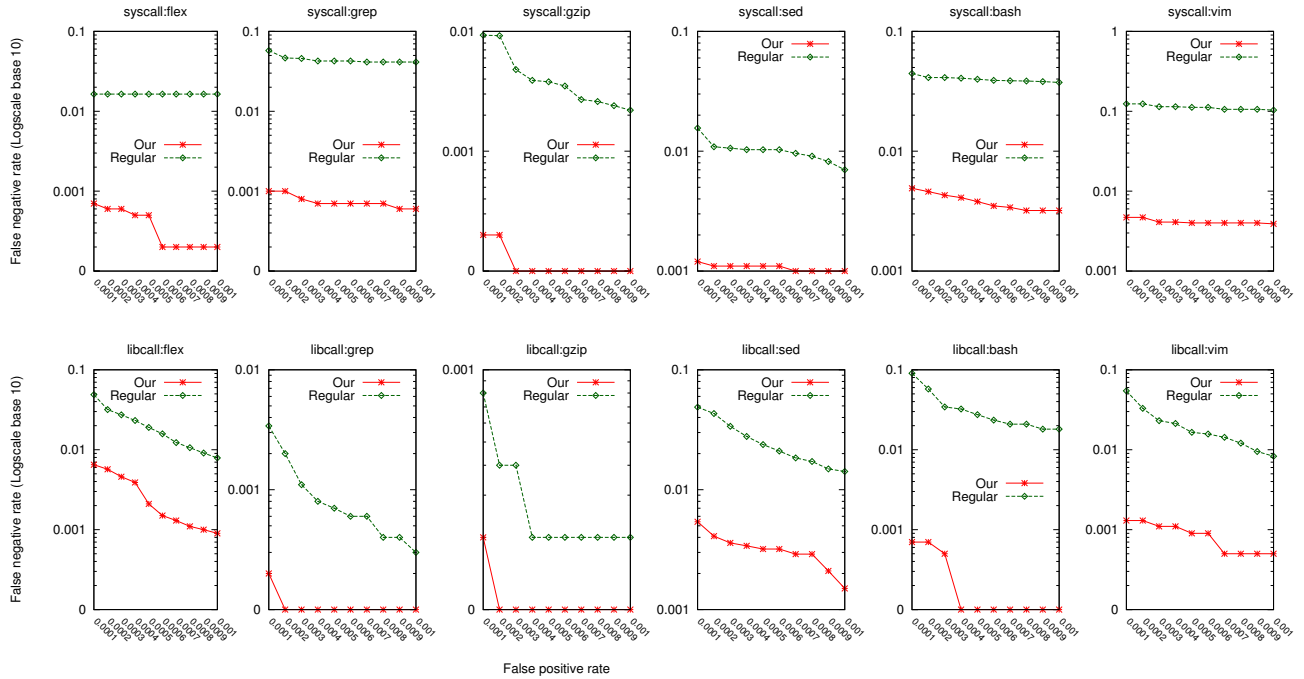


Figure 4.10: Comparisons of classification accuracy in our model and the regular HMM for library and system calls. X-axis shows false positive rates (misclassified normal segments). Y-axis (logscale, base 10) shows false negative rates (misclassified abnormal segments).

system calls available in the system, in application code, as well as in the observed traces of our experiments. So, system call sequences are less diverse than library call sequences in terms of their patterns. As a result, there are more normal system call segments with similar probabilities, which in turn leads to similar thresholds and similar corresponding false negative rates. This eventually makes the lines for system calls look more flat in general.

- At the same time, the characteristics of each application also contribute to the different looks of the figure lines for different applications, depending on the diversity and distribution of their call segment patterns.
- For one application, the choice of model is also a factor, a more accurate model can better distinguish different call segments even when they are similar, thus can give a line with more visible decrease along higher false positive. One example is the comparison between our model and random models for program flex on system call in Figure 4.10.

HMM models may give relatively high probabilities on a few abnormal traces which causes false negatives. The reason we think is that the abnormal trace segments are generated by replacing part of normal call segments with legitimate calls. These synthetic segments,

although different from the original normal traces, may still bear legitimate call transition patterns from the normal traces and thus obtain high probabilities. One way to improve is to use higher order HMM, which is able to capture longer call transition patterns.

4.3.5 Impact of Hidden States on Detection

The choice of initial parameters of a HMM is important in obtaining a more accurate model. Our approach initialize the HMM parameters such as number of hidden states, transition and emission probabilities, based on the information extracted from static analysis. These results indicate the importance of appropriate choice of initial model parameters, which is enabled by our static program analysis and model initialization process.

We compare the numbers of hidden states in four different models on library and system calls, also between our and regular models in Table 4.6. We also observe that STILO models have more hidden states than the regular models for both library and system calls (on average 0.1 to 2.3 times more).

Prog.	Number of hidden states used					
	libcall			syscall		
	n_o	n_r	$\frac{n_o}{n_r}$	n'_o	n'_r	$\frac{n'_o}{n'_r}$
flex	26	23	1.1	39	12	3.3
grep	49	40	1.2	35	12	2.9
gzip	45	36	1.3	40	17	2.4
sed	45	37	1.2	40	15	2.7
bash	129	86	1.5	66	43	1.5
vim	131	74	1.8	60	38	1.6

Table 4.6: Numbers of hidden states used in our HMM and the regular HMM for library or system calls. For regular model the number of hidden states used is the number of distinct calls in observed traces, n_r for libcall and n'_r for syscall. For our model it is the number of distinct calls in our aggregated call-transition matrix, n_o for libcall and n'_o for syscall.

We evaluate the regular models with different numbers N of hidden states, and compare their classification accuracy with STILO for all six programs from SIR at both syscall and libcall levels. For each setting, the execution is repeated three times and numbers are averaged.

The syscall results for **grep** and **gzip** are shown in Figure 4.11. We observe that:

- For the regular model, having more hidden states may or may not increase the model accuracy. Models with fewer hidden states sometimes outperform those with more states. (e.g. syscall for **grep**, **gzip**, **flex**, and **sed**).

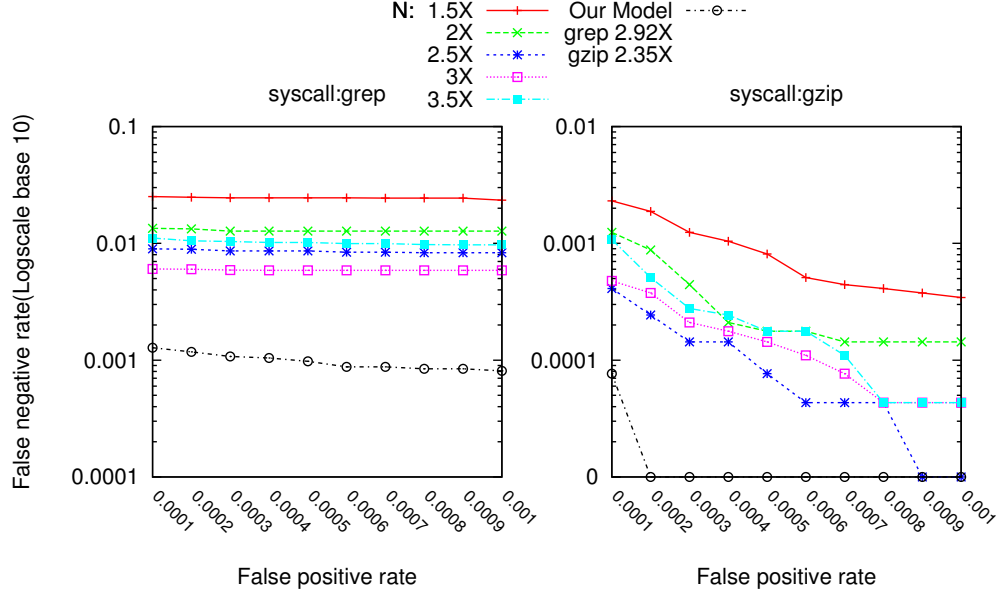


Figure 4.11: The (unpredictable) impact of the number N of hidden states on classification accuracy. Classification results of system calls by regular HMM with various numbers of hidden states for grep and gzip programs are shown, as well as results from STILO HMM. For each HMM, N is shown as multiples (X) of the number of distinct calls in traces.

- STILO consistently demonstrates higher classification accuracy, even when compared to regular models with more hidden states. This observation suggests our initial probability values also contribute to the improvement in classification performance.

For a regular model, parameter searching for the initial model is a time-consuming process. Using a large number of hidden states does not always guarantee a better classification in HMM. Training with many hidden states is time-consuming, as training a single trace in a single iteration takes $O(nm^2)$, where n is the length of a trace and m is the number of states. In comparison, STILO utilizes static program analysis results to effectively and efficiently provide the initial parameters of the behavior-learning model.

Detailed results are shown in Figure 4.12.

4.3.6 Detection of Code-Reuse Exploits

To demonstrate the ability to detect subtle code-reuse exploits, we reproduced both ROP and `return_to_libc` exploits [87, 88, 85, 86]. Some of these exploit and payload segments are entirely composed of existing legitimate calls in a new ordering. We also evaluated conventional code injection exploits. Details are shown in Table 4.7.

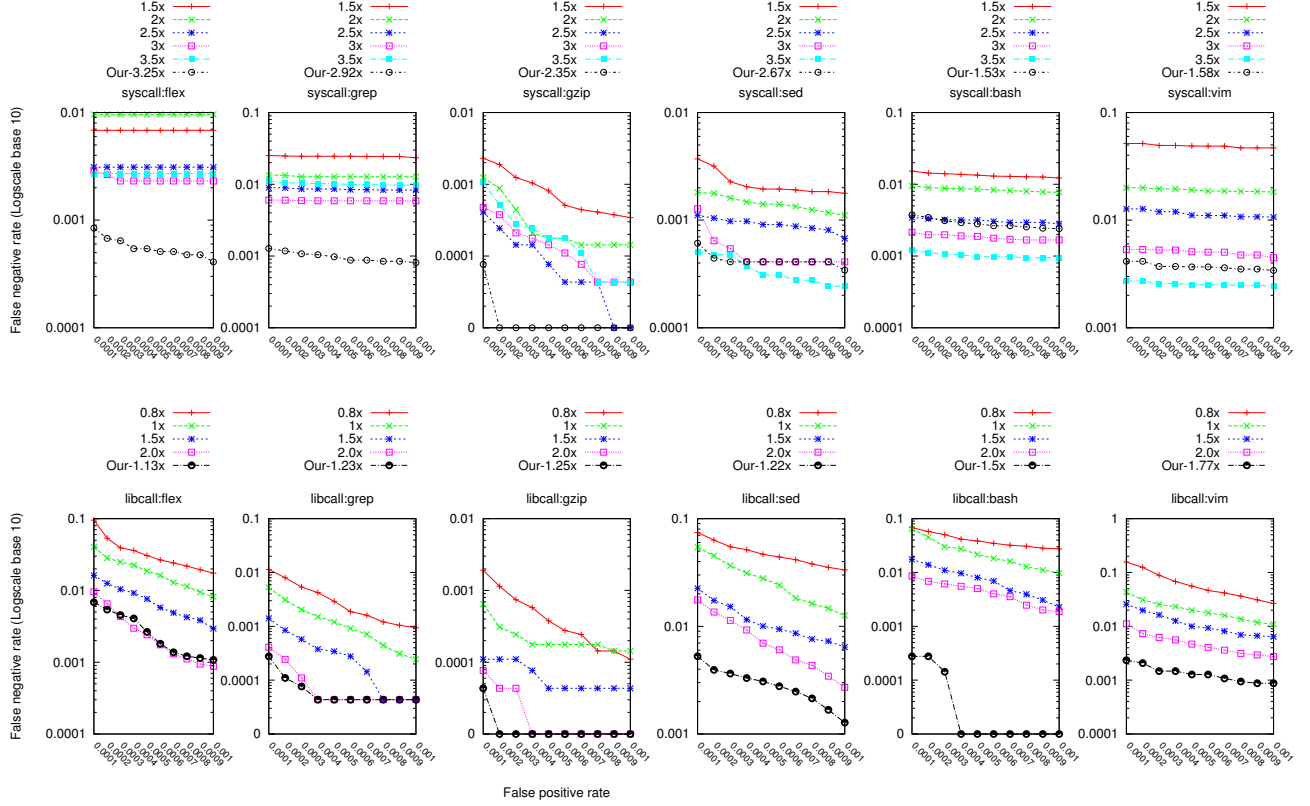


Figure 4.12: Comparison between our model and regular models with various numbers of hidden states (The legends in the figure indicate the number of states used, we use x to represent the number of states used in most basic regular model which is the number of distinct calls in trace). X-axis is the false positive rate (normal segments miss-classified), and Y-axis is the corresponding false negative rate (abnormal segments misclassified).

STILO successfully detects these exploits without triggering any false alerts in normal segments. It recognizes anomalous segments regardless of whether there are any new unseen system calls in them.

ROP-based syscall chains. We produced ROP-based syscall chains that allow attackers to create and execute a sequence of system calls using the instruction gadgets from the victim program `gzip`. A buffer overflow vulnerability was instrumented into `gzip`. We identified a few synthesized abnormal system call segments which are classified as abnormal by our model, but mis-classified as normal by the regular model. We reproduce the ROP shellcode which can make those identified system call segments. STILO successfully recognizes these system call chains as abnormal, i.e., these sequences generate zero or ultra low probabilities (e.g., $2.20 \times e^{-15}$) on the abnormal segments that are intercepted during the instruction-reuse exploit. In comparison, the regular HMM cannot recognize these segments as abnormal.

Vulnerability	Payload
Buffer Overflow (gzip)	<i>ROP_syscall_chain</i> <i>return_to_libc</i> <i>return_to_libc_chain</i>
Backdoor (proftpd)	<i>bind_perl</i> <i>bind_perl_ipv6</i> <i>generic cmd execution</i> <i>double reverse TCP</i> <i>reverse_perl</i> <i>reverse_perl_ssl</i> <i>reverse_ssl_double_telnet</i>
Buffer Overflow (proftpd)	<i>guess memory address</i>

Table 4.7: STILO successfully detects *Abnormal-A* segments from real-world exploits. It recognizes attack sequences without triggering any false alerts on *Normal* segments.

As an example, we list the classification probabilities by both our and regular models for such call segments. The false positive rate here is set to 0.0001, and the corresponding thresholds for STILO and regular models are shown in Table 4.8.

Segments	P (STILO)	P (Regular)
S_1	0.0	0.20
S_2	$2.20 \times e^{-15}$	0.29
S_3	$1.54 \times e^{-5}$	0.25
S_4	0.0	0.27
S_5	0.0005	0.33
S_6	0.0	0.23
S_7	0.0004	0.26

Table 4.8: STILO model generates zero or ultra low probabilities on abnormal segments during a subtle instruction-reuse ROP exploit. In comparison, these segments are not recognized by the regular HMM as abnormal (false negatives).

- S_1 : [uname, brk, brk, brk, brk, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, getdents, rt_sigaction, chown, chmod, exit_group]
- S_2 : [uname, brk, brk, brk, brk, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, fstat, mmap, write, unlink, chown]
- S_3 : [brk, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, read, read, write, utime, chmod, close, unlink, getdents]

- S_4 : [uname, brk, brk, brk, brk, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, getdents, openat, fstat, read, read]
- S_5 : [rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, stat, open, stat, open, fstat, stat, write, write, utime, unlink, chmod]
- S_6 : [brk, brk, brk, brk, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, chown, rt_sigaction, chown, chown, getdents]
- S_7 : [uname, brk, brk, brk, brk, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, read, read, getdents, open, write]

Return_to_libc. STILO can also detect several libcall-based code-reuse exploits that target a vulnerable `gzip`. `return_to_libc`'s payload uses `system()` libc function to open a shell. `return_to_libc_chain`'s payload invokes sequences of libc function calls to implement the `download_and_execute` action (`system()`->`setuid()`->`chmod()`->`execl()`). Another ROP exploit's payload executes shell commands attempting to steal sensitive information from the victim host, such as existing user name (by accessing `/etc/passwd`), the information of running processes (by calling `ps`). STILO detects all these attack call traces.

Backdoor. For `proftpd` server, we reproduced a backdoor trojan (OSVDB-69562) and a buffer overflow (CVE-2010-4221) exploit, and collect server-side traces as the result of exploits and attacker payloads. We compute the probabilities of observed call segments using trained HMM, and any previously unseen syscall or low probability below threshold is considered as an anomaly.

The backdoor vulnerability was found in a `proftpd` downloadable archive, allowing attackers to gain the privilege of remote command execution. In the buffer overflow exploit, an attacker attempts to guess memory offsets of instructions under ASLR through telnet connections. All the payloads used in the backdoor exploit are for establishing various types of communication channels (telnet, IPv6, oneway, bidirectional, TCP, or SSL) between the victim machine and the remote attacker.

Two examples of attack system call segments are:

- [read, read, close, munmap, stat, open, fstat, mmap, read, read, close, munmap, uname, socket, connect]
- [open, fstat, mmap, close, ioctl, ioctl, ioctl, rt_sigaction, execve, execve, execve, execve, brk, access, mmap]

In general, our model can detect any abnormal call sequence behaviors that do not match the trained hidden Markov model.

4.3.7 Running Time Information

Runtime of STATIC CFG CONSTRUCTION, PROBABILITY ESTIMATION, and AGGREGATION OF CALL TRANSITION MATRIX operations is shown in Table 4.9. Our processing with static analysis for model initialization is very efficient and only takes seconds to finish. The aggregation for `vim` takes more time than others, which we speculate is due to more function calls inside the program, thus more aggregation operations among functions.

Prog.	Time (lib) Time (sys)	CFG	Prob. Est.	Aggr.
flex		0.06	0.24	0.31
		0.51	2.67	4.08
grep		0.07	0.39	0.3
		0.51	2.76	4.01
gzip		0.04	0.08	0.28
		0.49	2.41	3.97
sed		0.08	0.15	0.55
		0.54	2.56	4.52
bash		0.46	1.11	9.43
		1.06	3.66	19.62
vim		0.65	2.48	218.04
		1.21	4.99	175.80
nginx		0.39	0.75	1.24
		2.45	8.29	41.06
proftpd		1.01	1.87	14.96
		3.01	9.39	55.78

Table 4.9: STILO analysis runtime in seconds. CFG is for CFG constructions. Prob. Est. is for probability estimation in functions. Aggr. is for the aggregation of call-transition matrices.

Figure 4.13 compares the number of iterations needed for HMM to converge for library calls and system calls in STILO and regular models. STILO models achieve significant speedup in the training iterations compared to regular models for both system calls and library calls (with length 15). Analysis on libcalls achieve a higher speedup than the analysis on syscalls, about 50% shorter for libcalls and 25% shorter for syscalls on average. Library calls, with large numbers of distinct calls, take more rounds to converge than system calls.

STILO HMMs take fewer iterations to converge, despite having more states. This observation indicates that our initial STILO HMM is closer to its optimum than the regular HMM, confirming the positive impact of our initialization.

Deployment in Production

For deployment, the audit framework can be used in production system to monitor programs

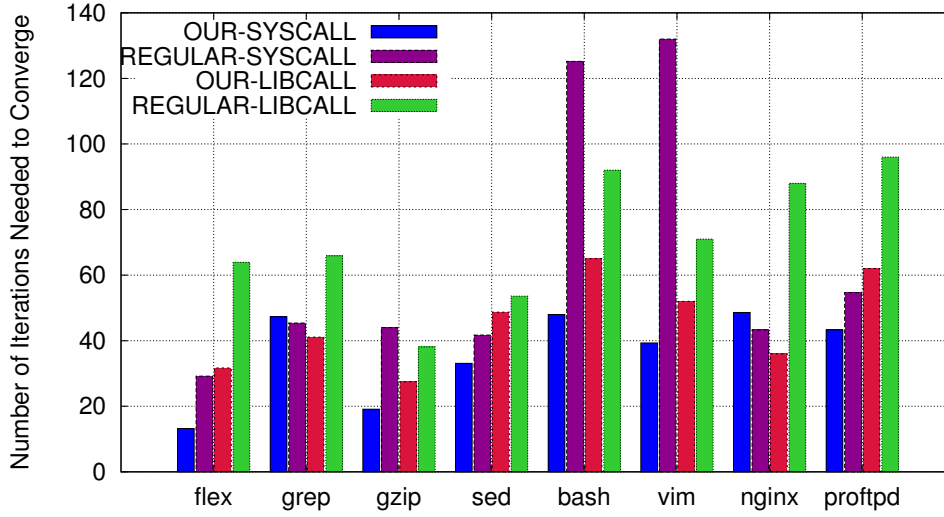


Figure 4.13: Numbers of iterations needed for HMM models to converge for library calls and system calls in our models and regular models. STILO models require much less training iterations than regular models.

and collect traces, with minimal impact on performance [4, 5]. The classifier component can work separately from the monitoring component, so it won't affect the normal execution of an application. The size of our model is within limited size, so the computation needed for classifying each call segment should also be limited. As an example, for the system call model of gzip, the average time on computing the probability of a 15-call segment is 0.03758519018 mili-seconds. Also our current HMM implementation is on JAVA, and can be further improved with c/c++.

4.3.8 Summary of Experimental Findings

Our experimental findings positively confirm our hypothesis that control-flow information extracted from static program analysis can significantly boost the classification accuracy in HMM-based anomaly detection techniques. We summarize our experimental findings below.

1. **The average classification accuracy of our STILO HMM is 11- to 28-fold higher than the hidden Markov models used by existing anomaly detection systems.** This trend is consistently observed in all the utility programs for both library calls and system calls as shown (in Figure 4.9). The high classification accuracy in STILO suggests the effectiveness of our static program analysis guided HMM initialization in boosting its security performance.

STILO HMM takes a fewer number of rounds (39% fewer on average) to reach convergence than the regular HMM. The result is shown in Figure 4.13.

This result shows that our initialization method facilitates the convergence of the hidden Markov model.

2. **STILO can outperform the regular HMMs with similar or more hidden states, suggesting the significance of our probability forecast in boosting detection accuracy.** A higher number of hidden states may or may not increase the classification accuracy, as shown in Figure 4.12. Therefore, we attribute STILO’s accuracy improvement to two reasons: **i)** an informed set of initial probabilities (transition and emission probabilities and initial probability distribution of hidden states) and **ii)** a more optimized number of hidden states.
3. We compare the impact of different segment lengths in model accuracy and show that longer segment is preferred.
4. **STILO detects all the library-call and system-call based code-reuse attacks evaluated, while maintaining zero false positive rates for normal call segments.** The attacks include `return_to_libc` and `return-oriented-programming` (ROP). STILO detects subtle code-reuse based anomalous sequences that are composed of legitimate call elements, whereas the regular HMM model cannot.
5. **Most STILO operations can be finished in seconds for the programs evaluated.** The runtime information of STILO’s analysis operations for library calls and system calls is shown in Table 4.9, including for STATIC CFG CONSTRUCTION, PROBABILITY ESTIMATION, and AGGREGATION OF CALL-TRANSITION MATRIX.
6. **Detection with library calls yield more precise results than that with system calls on average.** Classification accuracy based on libcalls is on average twice as high as that of syscalls. This trend is generally observed for both our model and the regular HMM with a few exceptions (Figure 4.9). Both types of call sequences reflect the control flow of program execution. We partially attribute the higher accuracy of using libcalls to the larger set of distinct calls as compared to syscalls, which results in a finer-grained representation of the program control-flow patterns.

Algorithm 3 Function for aggregating callee function's transition matrix P into caller function.

Input: Caller function g and callee function f 's call-transition matrices $g.P$ and $f.P$.

Output: The aggregated call-transition matrix $g.P$.

```

function AGGREGATE( $g.P, f.P$ )
  //handling  $g$ 's call to  $f$ 
  for all  $c_k \in f.P.callset \wedge f.P[\epsilon][c_k] \neq 0$  do //  $g$  calls into  $f$ 
    if  $c_k \notin g.P.callset$  then
      for all  $c_i \in g.P.callset \wedge g.P[c_i][f] \neq 0$  do
         $g.P[c_i][c_k] = g.P[c_i][f] * f.P[\epsilon][c_k]$ 
      end for
    else
      for all  $c_i \in g.P.callset \wedge g.P[c_i][f] \neq 0$  do
         $g.P[c_i][c_k] += g.P[c_i][f] * f.P[\epsilon][c_k]$ 
      end for
    end if
  end for
  //handling  $f$ 's return to  $g$ 
  for all  $c_l \in f.P.callset \wedge f.P[c_l][\epsilon'] \neq 0$  do
    if  $c_l \notin g.P.callset$  then
      for all  $c_j \in g.P.callset \wedge g.P[f][c_j] \neq 0$  do
         $g.P[c_l][c_j] = g.P[f][c_j] * f.P[c_l][\epsilon']$ 
      end for
    else
      for all  $c_j \in g.P.callset \wedge g.P[f][c_j] \neq 0$  do
         $g.P[c_l][c_j] += g.P[f][c_j] * f.P[c_l][\epsilon']$ 
      end for
    end if
  end for
  //handling call transitions inside  $f$ 
  for all  $(c_k, c_l) \in f.P.callset \wedge f.P[c_k][c_l] \neq 0$  do
    if  $c_k \notin g.P.callset \vee c_l \notin g.P.callset$  then
       $g.P[c_k][c_l] = \sum_i g.P[c_i][f] * f.P[c_k][c_l]$ 
    else
       $g.P[c_k][c_l] += \sum_i g.P[c_i][f] * f.P[c_k][c_l]$ 
    end if
  end for
  //when  $f$  does not make calls
  for all  $(c_i, c_j) \in g.P.callset$  do
     $g.P[c_i][c_j] += g.P[c_i][f] * f.P[\epsilon][\epsilon'] * g.P[f][c_j]$ 
  end for
  //remove  $f$  from  $g$ 's matrix
   $g.P.callset = g.P.callset + f.P.callset - \{f\}$ 
  return  $g.P$ 
end function

```

Chapter 5

Quantitative Call Sequence Modeling with Context Sensitivity

Our STILO-basic models in Chapter 4 is flow-sensitive, which detect program anomalies by characterizing the execution order and probabilities of system/library call sequences. In this chapter we aim at extending the program behavior models to be not only flow-sensitive, but also context-sensitive. The additional context information allows us to be certain about from where exactly each call is made, thus more capable to identify more sophisticated attacks.

5.1 Context-Sensitive Model Overview

Our proposed method combines the advantages of both dynamic learning and static analysis to construct a quantitative and also context-sensitive model for characterizing program behaviors. It associates context information to each system or library call to increase the precision of a detection model. In this section, we start with a small example to demonstrate the increased precision brought by modeling call sequences with context information.

Models that are context-insensitive record each library or system call by its call name, which is also the observation used in our basic flow-sensitive STILO model. Each of the call observations c in a context-insensitive model is represented as $c=\langle \text{call_name} \rangle$. Context-insensitive models rely on flow-sensitivity to capture the order and frequency of normal call sequences, and detect anomalies by identifying and classifying call sequence patterns.

Our new method in this chapter is not only flow-sensitive, but also context-sensitive. It associates the caller function of each library or system call as the context information. As a result, each call observation c carries more information, and can be represented as $c=\langle \text{call_name@caller_function} \rangle$. the caller function of each library or system call as the context information. Our exmaple shows the effectiveness of context information in anomaly

deteciton. We describe with the control-flow graphs for function f and g in Figure 5.1 and Figure 5.2, where g calls f .

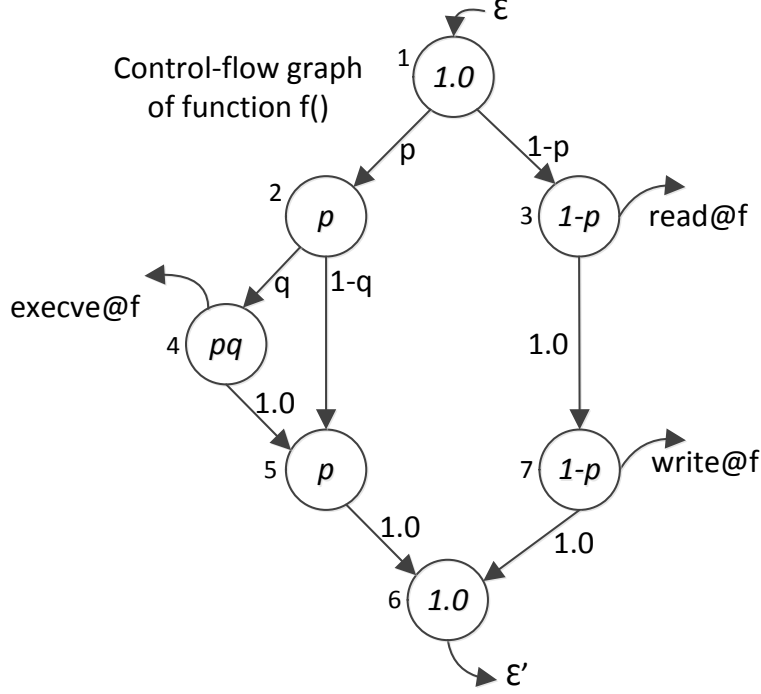


Figure 5.1: Examples of conditional probabilities and reachability probabilities for function $f()$. Conditional probability of a node pair is shown on the edge. Reachability probability of a node is shown in the node. ϵ and ϵ' represent the external call site and return site of $f()$.

In Figure 5.3, $S1$ gives a normal call sequence with the right context information, which is their caller function g and f . Suppose the corresponding program has a vulnerability and fails to check the boundary before the `read` call in function f , which would lead to a buffer overflow. An exploitation uses this security flaw and is able to launch a code-reuse attack which intentionally makes the same call sequence to avoid detection, but does malicious activities.

For a detection model that is flow-sensitive only, both normal call sequence and abnormal call sequence are observed the same as `read->read->write->execve`, thus the anomaly is not detected. Since code-reuse attack makes use of the existing code inside the entire process memory, it is common that an attacker uses calls that are made from different places within different caller functions. Although still conforming to normal call sequence order, the incorrect caller information as in $S2$ can be easily identified by a context-sensitive model.

Our approach adds the caller context to library and system calls, as they are the operations an attacker is more likely to make use of in order to conduct malicious activities. Adding

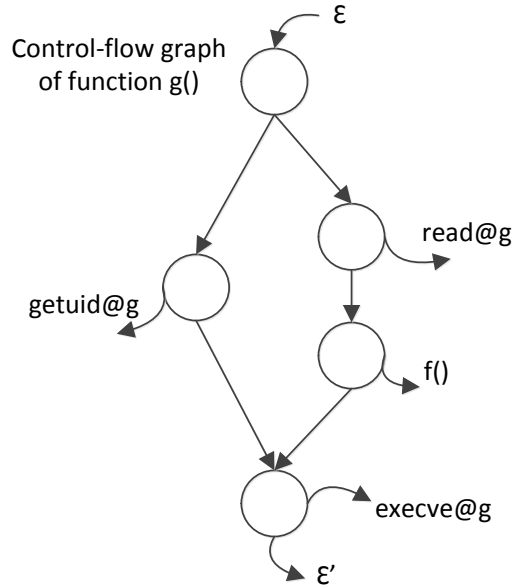


Figure 5.2: Example control flow graph for function $g()$, which is a caller function of $f()$.

S1: normal call sequence:

- ...->read@g -> read@f -> write@f -> execve@g->...

S2: abnormal call sequence (Code reuse attack):

- ...->read@g -> read@f -> write@foo -> execve@bar->...

Figure 5.3: Example of a normal and abnormal call sequences under a code-reuse attack. Caller information is associated to each call as the context information.

context information to other internal functions of a program can lead to a more precise representation of the program. However, this will incur heavy overhead during both analysis and runtime monitoring. In our model, we focus on characterizing the library and system calls, which are more security-related.

The integration of static analysis and dynamic learning improves flow-sensitivity, and enables our model to make quantitative classification with more complete coverage of call sequences. Adding caller information to each call observation further increases the precision of our model at a different dimension, i.e., context sensitivity.

Our program behavior-based detection model is constructed with the guidance of program analysis. With the capability to reason on occurrence likelihoods, our model is able to detect more stealthy attack sequences. Including context information allows the model to distinguish abnormal call sequences with increased precision. We give an overview of our

workflow below.

1. **Context Identification:** We parse the control-flow graphs of all functions and find the system and library calls inside them. Each identified system or library call is assigned additional context information to associate with its call name. This context information is maintained throughout our analysis to distinguish calls made at different functions.
2. **Probability Forecast:** Probability information is extracted from control-flow graphs to statically estimate the likelihoods of occurrence for call sequences. This probability forecast consists of two steps.
 - *Step 1:* The control-flow graph of a function is processed to output a *call-transition matrix* for this function (Definition 8). This matrix consists of estimated call transition probabilities, which represent the likelihoods of occurrence for sequences of calls when the corresponding function is executed.
 - *Step 2:* Individual transition matrices of functions are aggregated into one (larger) matrix to obtain the call transitions of the entire program. The aggregation of probability values are performed according to the call relations between the caller and callee functions based on the call graph.
3. **HMM Init:** The call-transition matrix of the program is used to initialize the parameters of a machine learning model (namely hidden Markov model). The parameters include the number of hidden states N , the collection of observation symbols and its number M , emission probability distribution matrix B representing likelihoods of emitting observation symbols by hidden states, transition probability A among hidden states, and the initial probability distribution π for hidden states.
4. **Train and Classify:** We further train the initialized model with normal program traces to adjust the parameters of the HMM learner, so that it can recognize dynamic code behaviors. When used for classification, given a segment of program traces (in system call or library call), the model computes the probability of the call segment. This probability is the summation over all possible hidden state sequences (using the forward algorithm) in the HMM. The classification decision is made with respect to a pre-defined threshold T on the production probability of a call sequence.

Our model is both flow-sensitive and context-sensitive, as the Markov model captures the order of execution of statements in the program, and context information distinguishes system or library calls which may have the same call name but are made at different places in a program.

The definitions of probabilities can be found in Section 4.1 of Chapter 4, which are used in our static analysis operations. The definitions include the conditional probability of adjacent

CFG nodes, the reachability probability from the function entry, and transition probability for a call pair.

These defined probabilities are used as follows:

1. Our method first traverses the control-flow graph of each function to statically approximate the *conditional probability* P_{ij}^c for each pair of adjacent nodes (such as $n_1 \rightarrow n_2$ in Figure 5.1).
2. Then, based on conditional probabilities, we compute the *reachability probability* P_i^r for each node n_i , which represents the likelihood of n_i being executed in the function.
3. Finally, with these reachability probabilities, we compute *transition probabilities* for call pairs within each function. The transition probabilities are further used to construct a call-transition matrix for each corresponding function.

Our approach monitors and enforces a program’s behaviors embodied as system and library call sequence patterns. We aim at detecting attacks that violate a program’s normal control-flow executions. Such violations are common among both *i*) conventional code-injection shellcode following a memory corruption or Trojan horse and so on, as well as *ii*) more subtle code-reuse attacks (such as ROP, `Return_to_libc`). In general, any uncommon execution of a program with altered control-flow can be potentially detected by our model. Although our model is not specifically designed to detect general mimicry attacks (which is an open problem), it can catch mimicries that involve the invocation of legitimate-yet-rare calls or paths having low likelihoods of occurrences. The quantitative measurement together with context-sensitivity makes it difficult for an attacker to develop an effective mimicry attack call sequence.

5.2 Probability Forecast of Context-Sensitive Call Sequences

5.2.1 Operations of Probability Processing

In this section, we describe the operations for realizing CFG-based control-flow probability forecast, and how context information is maintained and used during the process.

Reachability Probability:

Reachability probability inside each function’s control-flow graph is the basis of our probability forecast. To compute *reachability probability* (see Definition 6 in Section 4.2.1 of Chapter 4), our algorithm traverses a CFG and estimates the probability to reach a CFG node from the function entry.

Call-Transition Matrix of Function:

Having the reachability probabilities, we can compute likelihoods of occurrence for call pairs in a function, i.e., *transition probability* (see Definition 7 in Section 4.2.2 of Chapter 4). In a function $f()$ The call-transition probability is defined for a call pair (c_1, c_2) , where c_1 precedes c_2 . As a context-sensitive model, each call $c = \langle \text{call_name@caller_function} \rangle$

Table 5.1 shows the call-transition matrix of the function in Figure 5.1 under the uniform distribution for conditional probabilities (i.e., $p = q = 0.5$). This is the assumption used in our current model.

	ϵ'	read@f	write@f	execve@f
ϵ	0.25	0.5	0	0.25
read@f	0	0	0.5	0
write@f	0.5	0	0	0
execve@f	0.25	0	0	0

Table 5.1: A call-transition matrix of the function in Figure 5.1 under the uniform distribution for conditional probabilities. ϵ represents the external caller of this function. ϵ' represents the external return site. Each column or row represents a system or library call associated with its caller function as context information.

Aggregation of Call Transitions:

With the call transition matrices from each individual function, the AGGREGATION OF CALL TRANSITIONS operation further aggregate them into one monolithic matrix representation, which will be ready to integrate with the mathematical representation of a hidden Markov learning model.

The inputs for this operation include: **i)** the call graph of the program and **ii)** call-transition matrix for each function. As we aggregate callee functions' matrix representation into caller functions', the call graph is needed for the calling relations among functions.

We identify and compute new transition probabilities for the following cases of call pairs during the probability aggregation. Specifically, the corresponding call-transition matrix for caller function $g()$ is in Table 5.2. The aggregated call-transition matrix is in Table 5.3.

1. Call pairs like (c_{g_i}, c_{f_1}) are updated, which are the call transitions when $g()$ calls into $f()$. E.g., in Table 5.3, the transition probability for (read@g,execve@f) is 0.125.
2. Call pairs like (c_{f_n}, c_{g_j}) are updated, which are the call transitions when $f()$ returns to $g()$. E.g., in Table 5.3, the transition probability for (write@f,execve@g) is 0.25.
3. Call pairs made inside function $f()$ are updated, such as (c_{f_1}, c_{f_2}) , $(c_{f_{n-1}}, c_{f_n})$. E.g., in Table 5.3, the transition probability for (read@f,write@f) is 0.25.

4. Call pairs like (c_{g_i}, c_{g_j}) are updated, which are the call transitions when there is no call made in function $f()$. E.g., in Table 5.3, the transition probability for $(\text{read@g}, \text{execve@g})$ is 0.125.
5. Call pairs that are originally made in function $g()$ are updated, such as (c_{g_j}, c_{g_k}) . E.g., in Table 5.3, the transition probability for $(\text{getuid@g}, \text{execve@f})$ is 0.5.

Details of computation for different cases are described in pseudocode in Algorithm 3 in Section 4.2.3 of Chapter 4. The matrix output by AGGREGATION quantitatively represents the pair-wise control flow of the program obtained through the static program analysis.

	ϵ'	getuid@g	read@g	f	execve@g
ϵ	0	0.5	0.5	0	0
getuid@g	0	0	0	0	0.5
read@g	0	0	0	0.5	0
f	0	0	0	0	0.5
execve@g	1.0	0	0	0	0

Table 5.2: A call-transition matrix of the caller function in Figure 5.2 under the uniform distribution for conditional probabilities. ϵ represents the external caller of this function. ϵ' represents the external return site. Each column or row represents a system or library call associated with its caller function as context information.

	ϵ'	getuid@g	read@g	execve@g	execve@f	write@f	read@f
ϵ	0	0.5	0.5	0	0	0	0
getuid@g	0	0	0	0.5	0	0	0
read@g	0	0	0	0.125	0.125	0	0.25
execve@g	1.0	0	0	0	0	0	0
execve@f	0	0	0	0.125	0	0	0
write@f	0	0	0	0.25	0	0	0
read@f	0	0	0	0	0	0.25	0

Table 5.3: The call-transition matrix for caller function $g()$ after aggregation. Each column or row represents a system or library call associated with its caller function as context information.

During the aggregation operation, the context information of each call is unchanged. For example, the call `write@f` in the callee function f will continue to be represented as `write@f`, after the call is aggregated into the call transition matrix of f 's caller function g . Our model records the most immediate caller of each system call and library call as its context information, and this information is maintained throughout the static analysis including the aggregation of call transition matrices. Fixing context information at such granularity effectively enforces where a call can actually be made from, thus limits the flexibility an attacker may have during exploitation.

Summary In STILO-Context (STatically InitialLized markOv with context sensitivity), the analysis described in this section takes as inputs control flows that can be statically inferred, and transforms them into a rigorous probability representation for all the call transitions in the program. Context information, together with each system and library calls are preserved in the output. Program behaviors that are not covered by our static program analysis (e.g., function pointer, recursions and loops) will be learned from program traces by our STILO HMM model. We describe how STILO-context HMM utilizes the obtained probability values next.

5.2.2 HMM Initialization

Our hidden Markov model is initialized with the call-transition probabilities and call information obtained from the static program analysis. The initialized HMM still needs to be trained with program traces to learn and cover dynamic control-flow patterns. Our final HMM represents both static and dynamic prediction of the program’s behaviors.

Hidden state We give semantic meanings to the initial hidden states. We let them represent the logical reasons (or program phases) governing the actions of a program. In our prototype, for each distinct system call or library call `call@func` in the *aggregated call-transition matrix*, we assign a hidden state. Therefore, there is a one-to-one correlation between hidden states and the (context-sensitive) calls in the program. Two calls with the same name but different caller context are considered to be different calls, which is also the assumption during model initialization.

In our STILO prototype, the number N of hidden states is the total number of distinct calls in the program code. This design choice enables us to conveniently incorporate statically obtained information into HMM. In regular HMMs (e.g., [117, 124]), N is the approximated number of distinct calls in program traces.

As an alternative to the one-to-one correlation between hidden states and system or library calls, one can also use one hidden states to represent multiple system or library calls. The benefit of doing that is reduced size of the constructed hidden Markov model, which helps accelerating the model training process.

Observation symbol The observation symbols M need to be associated with observable program behaviors. We define the *observation symbols* as system calls or library calls.

Emission probability Because of the semantics of our hidden states, it is straightforward to initialize the emission probabilities. For each hidden state i , we assign a high emission probability (e.g., 0.5) for the call that i corresponds to, and assign random low probabilities to the rest of the observation symbols.

State-transition probability Our HMM’s state-transition probabilities $\{A\}$ are initialized with the transition probabilities $\{P_{ij}^t\}$ of call pairs in the program’s aggregated call-transition

matrix.

Initial probability distribution In STILO, because of our one-to-one correlation between hidden states and calls, the distribution π of hidden states is approximated based on the program’s call-transition matrix. Specifically π_i is initialized with the frequencies of call occurrences ($\sum_j P_{ij}^t$) and normalized.

5.3 Experimental Evaluation

We refer to our prototype *STILO*, short for **ST**atically **Initial**Lized mark**OV**. Our prototype for the static program analysis is implemented in C/C++ using the *Dyninst* library [29]. We use the system tools *strace* and *ltrace* to intercept system calls and library calls of running application processes as well as the instruction pointer at the time of each call. The instruction pointers are later translated to the caller function with the binary utility tool *addr2line*. The translation operation is efficient as results can be cached.

For performance consideration, alternative monitoring tools (e.g., auditd [4]) can be used by our implementation in production systems. An acceptable 10% overhead was reported on a hybrid benchmark with realistic workload for auditd [5]. The HMM training and evaluation code is written in Java using the *Jahmm* library [39]. The evaluation or classification of call sequence is relatively efficient. For example, the computation for a 15-call segment takes 0.038 milliseconds (STILO for gzip), also this operation can be done offline, and paralleled for accelerated processing.

For identifying system calls, we compile a program with static linking. The library calls of interest are the *glibc* library calls, which are a collection of C standard libraries.

5.3.1 Experiment Setup

The programs and test cases used in our experiments include utility applications (*flex*, *grep*, *gzip*, *sed*, *bash*, *vim*) from the Software-artifact Infrastructure Repository (SIR) [94], as well as a FTP server *proftpd* and an HTTP server *nginx*. These programs average over 52586 lines of code, and 1,139KB in size. The programs we tested include both utility applications and server programs, which are all potential victims of attacks such as memory corruption, back-door, or binary instrumentation/replacement by attackers. Coverages of the test cases for programs in SIR are summarized in Table 4.4 in Section 4.3.1 of Chapter 4.

We also tested server programs *proftpd* and *nginx* to get normal traces. For *proftpd*, we test it by connecting to the running server from a client, navigating around the server directories, creating new directories and files, downloading, uploading, and deleting files and folders. For *nginx*, our test cases include both static webpages and dynamic php webpages which interact with an SQL database we set up. Our test cases cover different media types

including text, images, scripts and video files with Flash and Mp4 formats. Normal `http` and encrypted `https` accesses are also tested.

Statistics of calls observed during execution and calls in the program code are shown in Tables 5.4 and 5.5.

Prog.	Num. of unique function calls	Num. of unique libcalls in static analysis	Num. of unique libcalls in traces	Num. of unique context-sensitive libcalls in static analysis	Num. of unique context-sensitive libcalls in traces
flex	201	26	23	100	79
grep	197	49	40	141	88
gzip	157	45	36	113	58
sed	308	45	37	215	116
bash	1591	129	86	1366	383
vim	1611	131	74	829	289
proftpd	1359	175	98	1115	295
nginx	1369	103	25	467	46

Table 5.4: Statistics of library calls in programs and traces.

Prog.	Num. of unique function calls	Num. of unique syscalls in static analysis	Num. of unique syscalls in traces	Num. of unique context-sensitive syscalls in static analysis	Num. of unique context-sensitive syscalls in traces
flex	1083	39	12	45	12
grep	1065	35	12	42	12
gzip	1047	40	17	47	17
sed	1194	40	15	46	15
bash	2546	66	43	73	43
vim	2631	60	38	66	38
proftpd	2682	92	50	98	50
nginx	7189	84	22	89	22

Table 5.5: Statistics of system calls in programs and traces.

We compare the classification performance of following models:

- *Regular-basic*: This is the widely accepted HMM-based classification, which is the state-of-the-art probabilistic anomaly detection model (e.g., [117, 124, 43]).

- *Regular-context*: Different from the Regular-basic model where each observation is a system or library call, the Regular-context model uses context-sensitive observations where each call is associated with its caller information.
- *STILO-basic*: Both Regular-basic and Regular-context construct their models by training with the execution traces only. The STILO-basic model is initialized with every step of our static analysis operations, only without the context information, i.e., only the call name is recorded for each system or library call observation. We call it STILO-basic model.
- *STILO-context*: In addition to STILO-basic, STILO-context builds its model with statically extracted context-sensitive call transition probabilities, and also goes through dynamic training with context-sensitive call traces.

We show through experiments how the integration of static analysis and context information help improve the model accuracy in anomaly detection.

The good coverage of test cases in SIR gives the regular HMM a fair chance in the comparison with our model, as the accuracy of a regular HMM relies heavily on completeness of training data. For the regular HMM, the set of observation symbols consists of distinct calls from execution traces. The number of hidden states is the size of the call set (i.e., the total number of distinct calls in the traces). The regular model randomly chooses the initial HMM parameters, including the initial transition probabilities, initial emission probabilities, and the initial distribution of hidden states.

Our experiments answer the following questions.

1. How much improvement in classification accuracy do STILO models provide compared to the regular HMM models? (In Section 5.3.2)
2. What are the reasons for STILO HMM improvement, and which type of models give more accurate classification, library calls or system calls? (In Section 5.3.2)
3. Can STILO detect real-world attack traces? (In Section 5.3.3)
4. How well does STILO-context model perform against advanced code-reuse attacks? (In Section 5.3.4)

Standard HMM procedures are followed for model training and testing. All comparable HMM models are subject to the same convergence criteria during training. To help avoid model overfitting, for both STILO and regular HMM models, 20% of the normal data is kept aside to determine the termination of training. After each round of training, the intermediate model is tested on the termination data set, and the training is stopped with a converged model when there's no significant improvement on the termination data set. We perform *10-fold cross validation* on the rest of the normal data.

Given a threshold T for a program, false negative (FN) and false positive (FP) rates in HMM are defined in Equations (5.1) and (5.2), where $\{S_A\}$ and $\{S_N\}$ denote the set of abnormal segments and the set of normal segments of the program, respectively, and P_{S_A} and P_{S_N} represent the probability of an abnormal segment and a normal segment, respectively.

$$FN = \frac{|\{S_A : P_{S_A} > T\}|}{|\{S_A\}|} \quad (5.1)$$

$$FP = \frac{|\{S_N : P_{S_N} < T\}|}{|\{S_N\}|} \quad (5.2)$$

Training and classification are on n -grams of program traces, where $n = 15$ in our experiments (i.e., all segments consist of 15 calls). We found that classification with segments of length 15 produces more precise results than shorter segments (e.g., length 9). Therefore, we use 15 as the segment length for our experiments. Duplicate segments are removed in our training datasets in order to avoid bias.¹

- *Normal* segments are obtained by running the target executable and recording the library call or system call segments as the result of the execution. A total of 130,940,213 such segments from around 4,000 test cases of eight programs are evaluated. A HMM classification model needs to give high probabilities to these normal sequences. The training of hidden Markov models requires normal sequences, not abnormal sequences. We test the trained models with two types of *abnormal call segments*. Those segments should be given 0 or low probabilities.
- *Abnormal-A* segments (or attack segments) are obtained by reproducing several real-world attack exploits and payloads.
- *Abnormal-S* segments (or synthetic abnormal segments) are generated by replacing the last 4 normal calls in a segment with randomly selected calls from the legitimate call set. The legitimate call set consists of the distinct calls in a program's traces. A total of 160,000 *Abnormal-S* segments are evaluated. Our use of *Abnormal-S* segments enables a rigorous and comprehensive accuracy assessment.

5.3.2 Classification Accuracy

For each program, we compare four different models including STILO-basic, STILO-context, Regular-basic and Regular-context models, in their abilities to recognize new normal segments that are separate from the training set through 10-fold cross validation with *Normal* segments. We also compare their abilities to recognize *Abnormal-S* segments.

¹Experiments were conducted on a Linux machine with Intel Core i7-3770 CPU (@3.40GHz) and 16G memory.

HMM computes the probability of occurrence for each call segment. The classification decision is made with respect to a probability threshold T . Different choices of T yield different false positive (FP) and false negative (FN) rates. We show the library call and system call results of server programs **proftpd** and **nginx** as an example in Figure 5.4 and Figure 5.5 respectively. In general, STILO-context models out-perform all other regular or context-insensitive models compared.

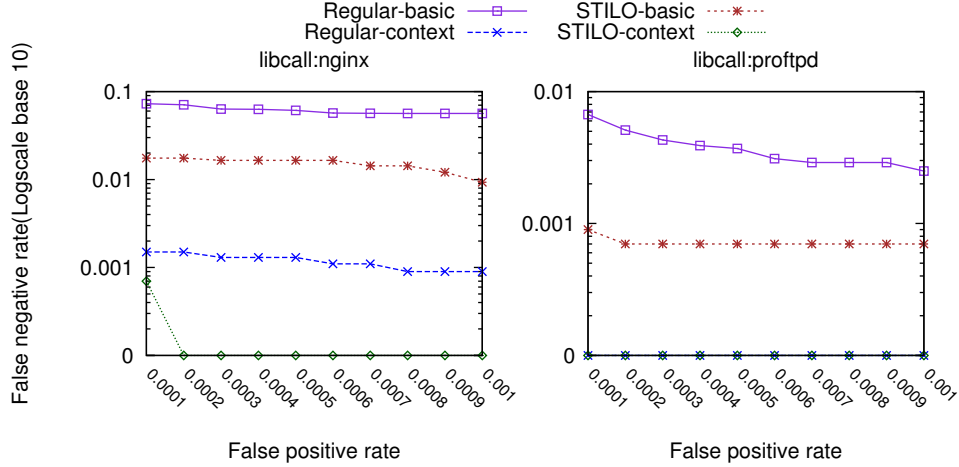


Figure 5.4: Comparison of STILO and Regular HMM's false negative rates (in Y-axis, base 10 log scale) for server programs **proftpd** and **nginx** on library calls under the same false positive rates (in X-axis). Clustering is applied to the initial model of **proftpd**.

Figure 5.6 shows the averaged false positive and false negative rates of STILO and the regular HMM for syscalls and libcalls. The average is taken over all the eight programs we've tested. The FP rates are in the X-axis. The FN rates in Y-axis are in base-10 log scale. Standard errors are shown as the whisker lines.

The accuracy improvement of STILO models over regular HMM models are in Table 5.6. For each model, we compute the average accuracy measurement (false negative rate) based Equation 5.3, where FN_{ij} is the corresponding false negative rate for program i at false positive j , and N the count of averaged numbers. The improvement is measured by multiples of one model over the other.

$$Average_FN = \frac{\sum_{Program_i} \sum_{FP_j} FN_{ij}}{N} \quad (5.3)$$

Library (libc) calls are usually directly used in user code by various internal functions of a program. As a result, the caller context distinguishes library calls within different functions very well. This leads to a relatively large number of distinct library calls in our constructed context-sensitive model as well as in the dynamic execution traces. With this great diversity

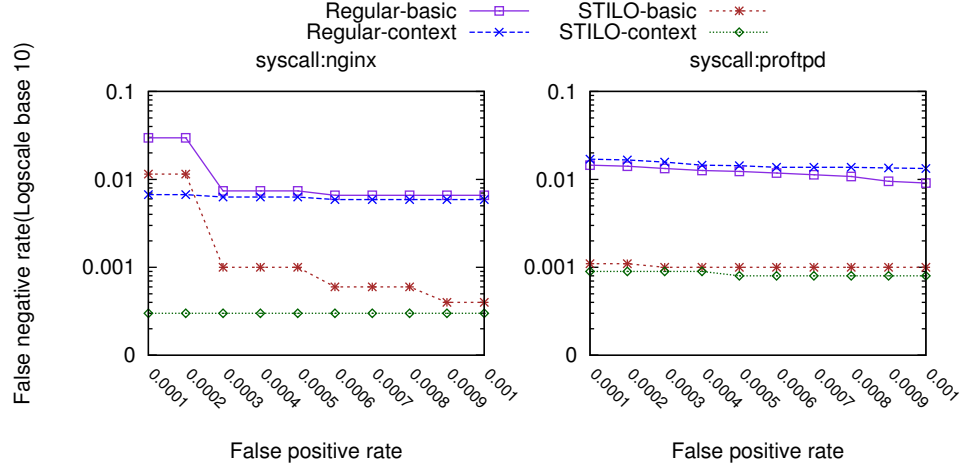


Figure 5.5: Comparison of STILO and Regular HMM's false negative rates (in Y-axis, base 10 log scale) for server programs **proftpd** and **nginx** on system calls under the same false positive rates (in X-axis).

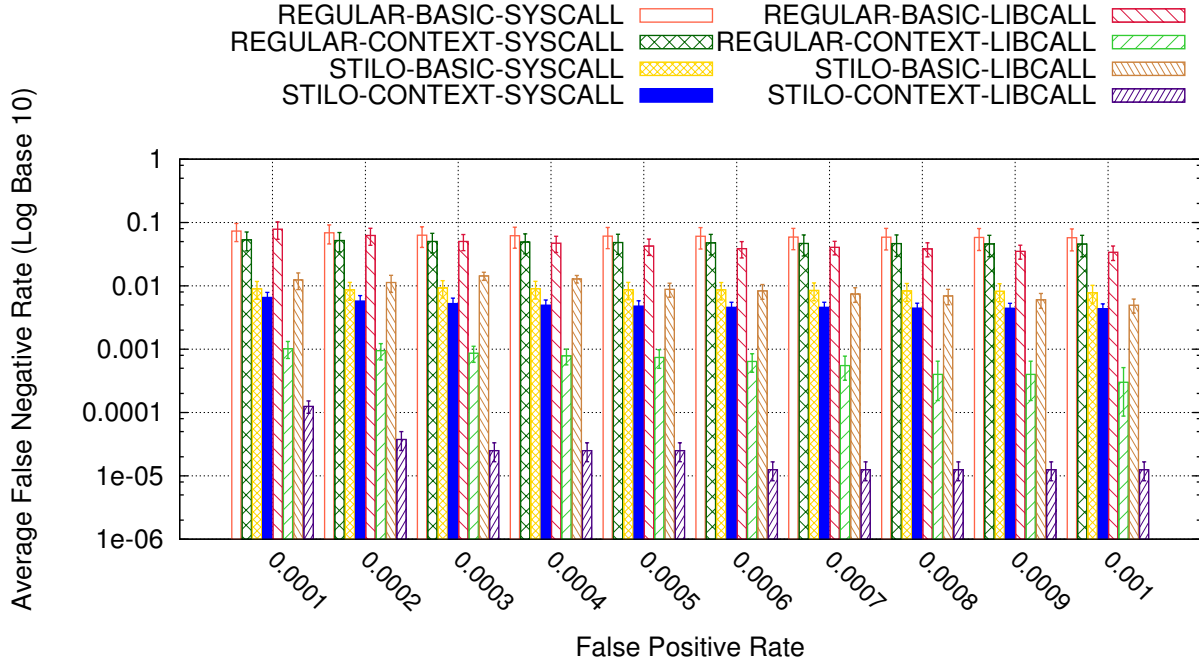


Figure 5.6: Comparison of the averaged false negative rates (in Y-axis) across the eight programs tested on system call or library calls by our basic and context-sensitive STILO models and the regular HMM, with respect to the false positive rates (in X-axis). Standard errors are shown.

Improvement (Row/Col)	regular-basic-syscall	regular-context-syscall	STILO-basic-syscall
regular-context-syscall	1.28	–	–
STILO-basic-syscall	7.24	–	–
STILO-context-syscall	12.61	9.87	1.75
Improvement (Row/Col)	regular-basic-libcall	regular-context-libcall	STILO-basic-libcall
regular-context-libcall	74.92	–	–
STILO-basic-libcall	5.23	–	–
STILO-context-libcall	2281.91	31.23	452.26

Table 5.6: Averaged accuracy improvement of STILO models over regular HMM models. The accuracy of each model is measured by its false negative rate averaged over tested programs and evaluated false positive rates, as in Equation 5.3.

of libc calls, context-sensitive models (including STILO-context and Regular-context) outperform STILO-basic and Regular-basic HMM models by significant margin, which is shown in Figure 5.4 and Figure 5.9.

For our STILO-context-libcall model, most of the cases the false negative rate drops to zero at the presented false positive rates, and this performance improvement is observed for almost every program tested in our experiment. The STILO-context-libcall models achieves false negative rates which are three orders of magnitude lower than the Regular-basic-libcall models. This improved model accuracy enables our model to better recognize various abnormal library call sequences.

System calls are often included in their corresponding wrapper functions, thus do not have great diversity in terms of their caller functions. In this case, the static analysis shows more impact on the accuracy of models where both STILO-context and STILO-basic models demonstrate lower false negative rates than the Regular-context and Regular-basic models, as in Figure 5.5 and Figure 5.8. Context-sensitive and context-insensitive models (Regular-basic and Regular-context, STILO-basic and STILO-context) usually have similar numbers of distinct system calls, thus similar numbers of states in the models, as a result their false negative lines are very close with small gap as shown in the figure.

On the normal and synthesized abnormal call segments, our STILO-context-syscall model is on average over 12 times better than the regular-basic-syscall model. The STILO-context-syscall places strict enforcement on how a system call is made by a program, as each system call can only be issued from very few legitimate caller functions. This restriction on caller context greatly limits an attacker’s degree of freedom within a victim application.

Table 5.7 gives the average number of iterations for a HMM model to converge. The number is averaged across all eight programs evaluated in our experiments.

Both our STILO-basic and STILO-context HMM models take fewer iterations than the regular HMM models to converge, despite having more states in the models. On average our STILO models takes 30% less iterations to converge than regular models. STILO-context HMMs take a little more iterations to converge than basic STILO HMMs, which we speculate is due to that the additional context information makes the initial STILO-context

Model	# of Iterations to Converge
regular-basic-syscall	64.44
regular-context-syscall	64.36
STILO-basic-syscall	36.49
STILO-context-syscall	39.59
regular-basic-libcall	71.06
regular-context-libcall	66.35
STILO-basic-libcall	45.50
STILO-context-libcall	64.94

Table 5.7: Average number of training iterations needed for an HMM to converge. Comparisons are among our basic and context-sensitive STILO models and regular HMMs.

HMMs more complex with bigger sizes. This observation indicates that our initial STILO HMMs is closer to its optimum than the regular HMMs, confirming the positive impact of our initialization. Detailed numbers for each of the tested programs are in Figure 5.7.

Clustering initial hidden states.

For models with large number of hidden states, model training can take a long time, as the time complexity of each training iteration is $O(TS^2)$, where T is the length of trace and S is the number of hidden states in the model.

To accelerate the training process, we applied clustering to cases where the simple one-to-one call-state mapping leads to a big initial model, and thus long training period. The transit-from (column) and transit-to (row) probabilities of each call in the aggregated call transition matrix is used as the representing vector, and K-means clustering is applied. Calls with similar probability vectors are assigned to one same hidden state to share the emission probability, and their transition probabilities are added up with same weight. Pseudocode of the operation is described in Algorithm 4.

The clustering operation is performed on the aggregated call transition matrix of a program. The semantic meaning behind the operation is the merging of similar calls, and the similarity of two calls is measured in two ways: i) the sets of incoming and outgoing calls, ii) the distribution of probabilities from the incoming and to the outgoing calls. From the perspective of program analysis, similar call transitions in CFGs and CGs of a program are represented together, in a compact way.

We applied the clustering to programs **bash**, **vim**, and **proftpd**, under STILO-context-libcall model. Table 5.8 shows the reduction of model sizes and estimated training speedup. Despite the reduction in model sizes, our STILO-context models still out-perform others as show in Figure 5.4 and Figure 5.9.

Possible HMM Overfitting:

Figure 5.8 and Figure 5.9 give the performance comparison among STILO and Regular HMM

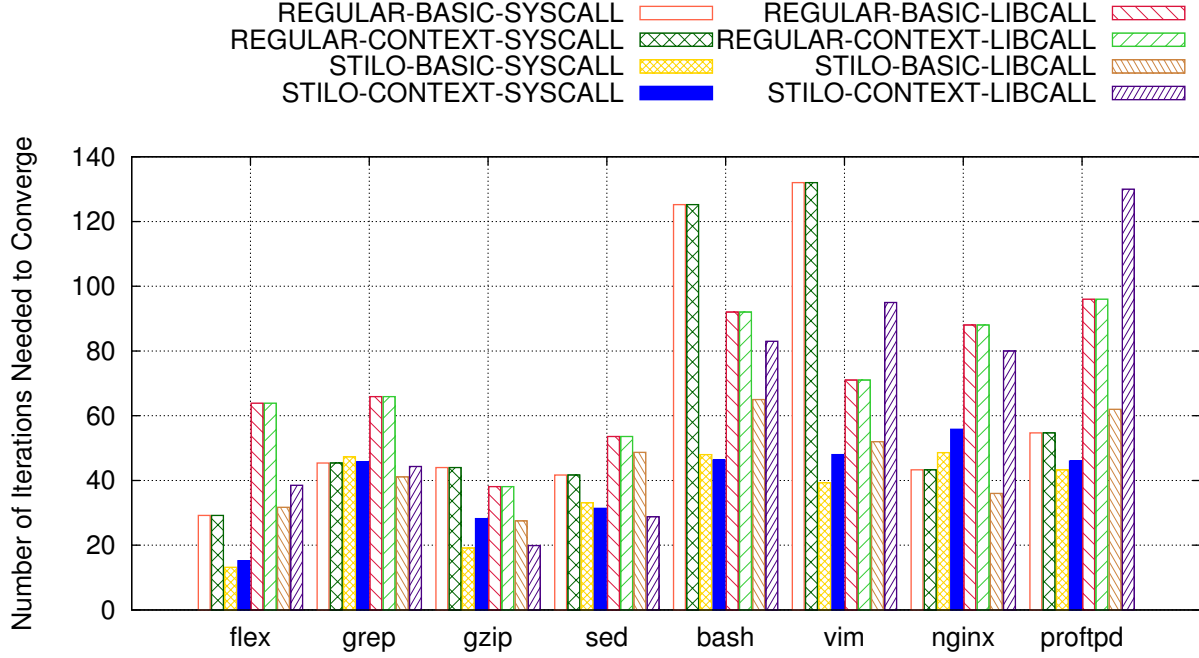


Figure 5.7: Numbers of iterations needed for HMM models to converge for library calls and system calls in our basic and context-sensitive STILO models and regular HMM model. STILO models require less training iterations than regular models.

Program	Model	# distinct calls	# states after clustering	Estimated training time reduction
bash	STILO-context-libcall	1366	455	88.91%
vim	STILO-context-libcall	829	415	74.94%
proftpd	STILO-context-libcall	1115	372	88.87%

Table 5.8: Statistics about the clustering of calls in the initial HMM models for selected programs and models and the estimated speedup.

models. In general, STILO models out-perform regular HMMs, and context-sensitive STILO models work better than basic STILO models with lower false negative rates.

One exception is for **flex** in system call case, where the context-sensitive STILO model shows higher false rates than the basic STILO model, which we discuss more below.

Among many potential factors which could have contributed to this exception, we explore the possibility of model overfitting. First of all, overfitting happens when a model is excessively complex. For example, a model with too many parameters will memorize the training data, instead of learn from it. Overfitting also depends on other things such as the conformability of the model structure with the data shape, and the magnitude of model error compared to the expected level of noise or error in the data. In order to avoid overfitting, the basis of some techniques is either (1) to explicitly penalize overly complex models, or (2) to test the model’s ability to generalize by evaluating its performance on a set of data not used

Algorithm 4 Pseudocode for Clustering Similar Calls of a Program Based on Call Transitions.

Input: The aggregated call transition matrix of a program.

Output: The clustered call transition matrix of the program.

```

function CLUSTER(matrix m)
  /* Each call is represented with a vector, which is the concatenation of a row of outgoing
  probabilities and a column of incoming probabilities. */
  for ( $i = 1 \rightarrow m.size()$ ) do
     $vectors[i] = (m[i], m^T[i])$ 
  end for
  /* Cluster vectors. */
   $clusters = cluster\_vectors(vectors)$ 

  /* Reconstruct clustered call transition matrix*/
  for ( $j = 1 \rightarrow clusters.size()$ ) do
     $n = clusters[j].size()$ 
    for  $index$  in  $clusters[j]$  do
       $m\_new[j][*] += m[index][*]/n$ 
       $m\_new[*][j] += m[*][index]/n$ 
    end for
  end for
   $return\ m\_new$ 
end function

```

for training, which is assumed to approximate the typical unseen data that a model will encounter.

In our experiment, to avoid overfitting during training, we use 20% of the trace as the termination set, while the rest 80% is used for 10-fold cross validation. After each round of training, the model is tested on the termination set, and the training will be stopped when there's not much improvement.

Having this separate termination set should have helped avoid overfitting, but it is not working for the case of program **flex**. We list the information of the termination set for each tested program in Table 5.9. **flex** has the least number of distinct call segments in the termination set, which means there's less diversity for call sequence patterns in its traces when compared with other programs. So it's possible that the context-sensitive model for **flex** is a bit overly complex, while at the same time, the small termination set is not able to stop the training in a timely way. And as a result, the context-sensitive STILO model, which has more states (thus more complex model), does not work as well as the basic STILO model.

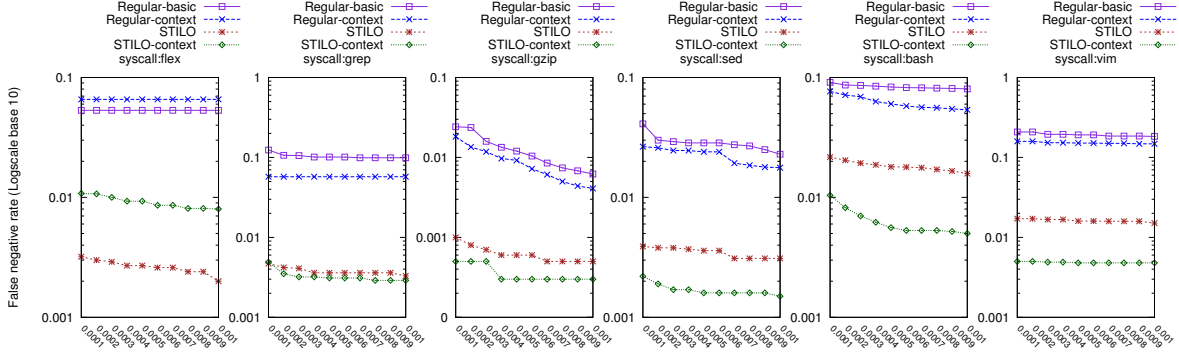


Figure 5.8: Comparison of STILO and Regular HMM’s false negative rates (in Y-axis, base 10 log scale) for evaluated utility programs on system calls under the same false positive rates (in X-axis).

Despite this exception, both our basic and context-sensitive STILO models out-performs regular HMM.

Program	# of distinct call segments in termination set
sed	162
gzip	63
grep	413
flex	34
bash	1274
vim	2008
nginx	341
proftpd	1037

Table 5.9: Statistics about termination set. The termination set is part of the traces which is kept aside and used during training to prevent model overfit.

5.3.3 Detection of Real-World Exploits

For the demonstration of our model’s ability to detect exploits in real attacks, we reproduced both conventional code injection exploits (including buffer overflow, backdoor trojan), as well as more advanced code-reuse attacks such as ROP. Details are shown in Table 5.10. With the enforcement of caller information, our context-sensitive models successfully detect all the attacks by identifying abnormal caller context (either missing or incorrect) of the observed call traces. By examining every call trace, high percentage of system calls (mostly ranging from 30% to 90%) were found with abnormal caller context information

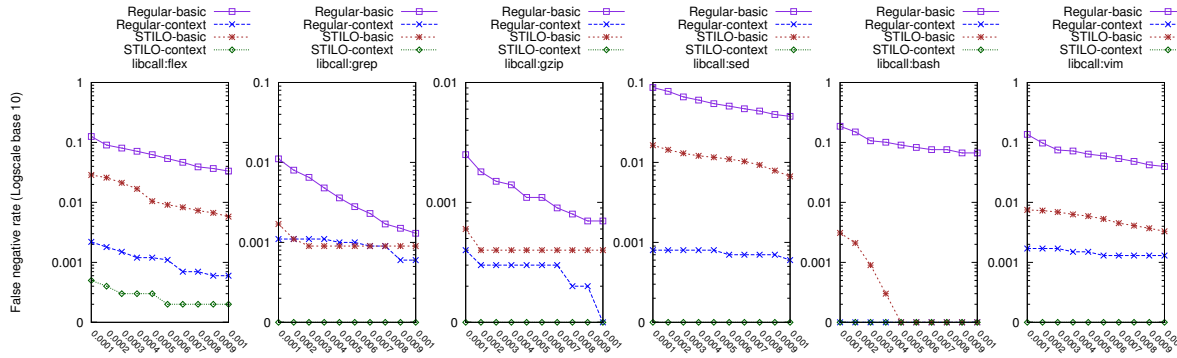


Figure 5.9: Comparison of STILO and Regular HMM’s false negative rates (in Y-axis, base 10 log scale) for evaluated utility programs on library calls under the same false positive rates (in X-axis). Clustering is applied to the initial models of **proftpd**, **vim** and **bash**.

Vulnerability	Payload	Detection
Buffer Overflow (gzip)	<i>ROP</i>	✓
	<i>ROP_syscall_chain</i>	✓
Backdoor (proftpd)	<i>bind_perl</i>	✓
	<i>bind_perl_ipv6</i>	✓
	<i>generic cmd execution</i>	✓
	<i>double reverse TCP</i>	✓
	<i>reverse_perl</i>	✓
	<i>reverse_perl_ssl</i>	✓
	<i>reverse_ssl_double_telnet</i>	✓
Buffer Overflow (proftpd)	<i>guess memory address</i>	✓

Table 5.10: STILO successfully detects *Abnormal-A* segments from real-world exploits.

Code injection attacks.

For **proftpd** server, we reproduced a backdoor trojan (OSVDB-69562) and a buffer overflow (CVE-2010-4221) exploit, and collect server-side traces as the result of exploits and attacker payloads. The backdoor vulnerability was found in a **proftpd** downloadable archive, allowing attackers to gain the privilege of remote command execution. In the buffer overflow exploit, an attacker attempts to guess memory offsets and call useful attack instructions under ASLR through telnet connections. All the payloads used in the backdoor exploit are for establishing various types of communication channels (telnet, IPv6, oneway, bidirectional, TCP, or SSL) between the victim machine and the remote attacker.

Code injection attacks execute payloads that are injected into the victim program’s memory. The payloads provide attackers the functions needed to conduct malicious activities. How-

ever, the illegal execution of external code makes system calls at places that are different from the normal execution, thus many of the calls in the traces are found missing caller context, and the anomaly is immediately detected. Table 5.10 shows that the *Abnormal-A* traces, which are from backdoor and buffer overflow attacks on `proftpd`, have high percentages of calls with abnormal caller context information.

Code-reuse attacks.

We reproduced return-oriented-programming attacks where an attacker can either execute an arbitrary command, or call a chain of system calls directly. Both cases use the instruction gadgets from the victim program `gzip`. A buffer overflow vulnerability was instrumented into `gzip`, which allows us to make further ROP executions.

Since a code-reuse attack makes calls using the existing code in the victim program, caller context information is found for each call. However, the caller context information and the actual call that is made from the context does not match with each other, because of the way ROP gadgets are used, which we discuss more in Section 5.3.4. This mismatch causes a high percentage of calls with abnormal caller context information, and leads to the detection of the attack.

With appropriate ROP gadgets, an attacker is able to make any call sequences wanted. If the attacker has knowledge about the detection model, s/he can for example further make a sequence of system calls which are carefully chosen to avoid the detection model, and at the same time still realize malicious actions. As an example, with ROP gadgets that exist in the binary of `gzip`, we reproduced the following two call segments *S1* and *S2* which are mistakenly considered as normal by the context-insensitive models, Regular-basic and STILO-basic respectively (at false positive rate 0.0001). Our STILO-context model immediately identifies the incorrect caller context and detects the anomaly.

- S1: [uname, brk, brk, brk, brk, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, read, close, close, unlink, chmod]
- S2: [brk, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, stat, openat, getdents, close, write, read, write, write]

5.3.4 ROP Difficulty Under Context-Sensitive Model

After a successful exploitation, the most common thing to follow is starting a terminal, which allows the attackers to perform further malicious activities. However, starting and using a new shell usually triggers a series of system calls, which are in an abnormal pattern and also from wrong caller context. This difference can be quickly identified by our detection model. As a result, to avoid detection and to make the attacks more stealthy, attackers would need to make system calls directly, instead of running a shell.

Max Length of Instruction Gadget	2	6	10
Program	# of Useful [SYSCALL...RET] Gadgets		
sed	5	6	7
gzip	5	6	7
grep	5	6	7
flex	5	6	7
bash	9	12	15
vim	6	7	8
proftpd	8	13	17
nginx	8	11	16
libc.so	8	14	19

Table 5.11: Search result of useful [SYSCALL...RET] gadgets for making a sequence of system calls. Search includes evaluated binaries and libc.

On a x86 64-bit system, where our experiments are conducted, the way to make a system call is to use the **SYSCALL** instruction [121]. Together with the correct register settings, execution is transferred to the corresponding kernel procedures by the **SYSCALL** instruction.

In return oriented programming, instruction sequences ended with a **RET** is considered as a useful gadget, because it allows the next gadget to be loaded and the execution to be continued. So in order to make a sequence of system calls, one need to find gadgets which start with **SYSCALL** instruction, and end with a **RET**.

To evaluate the feasibility of such an attack, we searched each of the binary files for gadgets meeting these requirements with ROP gadget search tool [89]. We set a maximum length of a gadget during the gadget searching, and max length of 2, 6 and 10 instructions are tested during the search. It's possible that one may find an even longer gadget that satisfies the requirements, however, the longer a gadget is, the more likely that instructions between **SYSCALL** and **RET** may interfere with the normal execution. For example, the **EAX** register, which is usually where the return value is stored, might be overwritten before **RET** and the execution of next gadget.

For statically linked binaries, each binary only has a few gadgets found that meet the requirements. The result of the search is in Table 5.11. Under a context-insensitive model, each of the gadgets can be used to make any system call, which allows an attacker to make arbitrary system call sequences. However, when considering the correctness of context information, each of the gadgets is limited to only make the system call that conforms to the corresponding caller context. For example, the **gzip** binary only has 5 useful gadgets when searching with max length 6, and the only system calls that conform to context and can be used by an attacker are **getegid**, **geteuid**, **getgid**, **getuid**, **time**, **futex**. For dynamically linked binaries, no such gadget is found within the programs themselves. We also searched the common shared library **libc.so** and found only 14 such gadgets when searching at max length 6.

The possible system calls include: `getuid`, `madvise`, `futex`, `geteuid`, `getppid`, `umask`, `close`, `chown`, `getpgrp`, `getegid`, `getgid`, `clock_gettime`, `error`. The limited number of available [SYSCALL...RET] gadgets greatly restricts the possible malicious activities an attacker is allowed to do. Together with flow-sensitivity, our context-sensitive detection model makes it difficult for an attack to conduct arbitrary malicious activities without being caught.

5.3.5 Summary of Experimental Findings

We summarize our experimental findings below.

1. **The average classification accuracy of our context-sensitive STILO-context models is orders of magnitudes higher than the regular hidden Markov models used by existing anomaly detection systems.** This improvement is consistently observed for all the tested utility and server programs on both library and system calls as shown (in Figure 5.6). The high classification accuracy in STILO-context model suggests the effectiveness of our static program analysis guided HMM initialization in boosting its performance for anomaly detection.
2. **STILO HMMs (STILO-basic and -context) take fewer number of rounds of training (30% fewer on average) to reach convergence than the regular HMMs (Regular-basic and -context).** The detailed results are in Table 5.7 and Figure 5.7. This results show that our initialization method places the initial HMMs at a better start and facilitates the convergence of the hidden Markov model.
3. **STILO-context model detects all the code-injection and subtle code-reuse attacks evaluated.** STILO-context also detects carefully prepared ROP-based anomalous system call sequences by identifying their incorrect caller context, whereas the regular HMM model cannot.
4. **Most STILO operations can be finished in seconds for the programs evaluated.** The runtime information of STILO’s analysis operations for library calls and system calls is shown in Table 5.12, including for STATIC CFG CONSTRUCTION, PROBABILITY ESTIMATION, and AGGREGATION OF CALL-TRANSITION MATRIX.
5. **Detection with library calls yield more precise results than that with system calls on synthetic abnormal call sequences.** This trend is generally observed for all four compared detection models with a few exceptions (Figure 5.6). Both types of call sequences reflect the control flow of program execution. We partially attribute the higher accuracy of using libcalls to the larger set of distinct calls as compared to syscalls, which results in a finer-grained representation of the program control-flow patterns.

6. **The system call based detection model defends well against ROP-based code-reuse attacks.** System calls usually have fixed wrapper functions as the caller context. By enforcing the correct caller function information, our STILO-context model greatly weakens the ability of an attacker by limiting the possible system calls s/he can make during an attack, as shown in Table 5.11.

Prog.	Time (lib) Time (sys)	CFG	Prob. Est.	Aggr.
flex		0.06	0.24	0.75
		0.51	2.67	10.94
grep		0.07	0.39	0.56
		0.51	2.76	10.56
gzip		0.04	0.08	0.65
		0.49	2.41	10.66
sed		0.08	0.15	1.97
		0.54	2.56	12.55
bash		0.46	1.11	134.93
		1.06	3.66	75.94
vim		0.65	2.48	1435.73
		1.21	4.99	736.79
nginx		0.39	0.75	1.60
		2.45	8.29	55.94
proftpd		1.01	1.87	17.22
		3.01	9.39	57.45

Table 5.12: Analysis runtime for STILO-context model in seconds. CFG is for CFG constructions. Prob. Est. is for probability estimation in functions. Aggr. is for the aggregation of call-transition matrices.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

For the detection of infection onset, we focus on the drive-by download attack, which is a major vector for malware infection. We illustrated the analysis of user behaviors for protecting the system integrity of a computer, in particular for detecting the onset of malware infection via drive-by download. We described the design, implementation, and use of *DeWare* for host-based security protection against unauthorized system events through enforcing dependency among events within the operating system environments. We realized DeWare through defining and enforcing access-control policies across multiple domains (i.e., file system, process management, and user inputs) within the operating-system environment. Our user-behavior based policies are new and we demonstrated their use in solving the practical DBD detection problem. DeWare can be easily deployed and used in Windows.

To the quantitative modeling of program behavior, we have achieved the program-behavior-modeling goals that are set for detecting control-flow anomalies: **probabilistic modeling**, covering both **static** and **dynamic** control-flow behaviors. The probabilistic program modeling at the control-flow level for anomaly detection is new. It substantially improves the coverage and granularity of the existing static or dynamic analysis based anomaly detection systems, enhancing the detection capability.

STILO achieves context-sensitivity by including the caller function of each library or system call, which limits the capability of code-reuse attacks, and also has a good balance between runtime performance and model precision. We provide a rigorous and general framework and algorithms for performing probability analysis on statically inferred control flows, and its seamless integration with a probabilistic learning model. Extensive experimental evaluation confirmed the advantages of STILO HMMs in distinguishing normal and abnormal traces of various kinds, when compared with the widely accepted HMM-based anomaly detection methodology.

6.2 Future Work

We envision to extend the work presented in this dissertation in two directions. First, tracking the origin and provenance of critical system events by enforcing the dependency between them and user actions is a novel and effective approach for managing a secure host. We envision that our system-level access control approach can be generalized beyond file and process events within OS environments. For example, the dependency between user actions and outbound network traffic can be identified and enforced for blocking stealthy malware in the future. We also see this be a promising approach in the research area of mobile security, as mobile devices naturally have richer and closer interactions from users, with the touch-oriented interface.

The second direction for future work, is to further extend our current hidden Markov model based approach:

- *Incremental training* Current model is obtained by training with a batch of normal data set. As we’ve already discussed in section 4.1, it happens that the training set could be incomplete. When later new normal data is observed, it needs to be added to the current trained model. Retraining a new model with all the previous data and new data is time-consuming, so we need the support of incremental learning [59] in STILO.
- *Higher-order Markov chain* Current solution assumes a 1st-order Markov model (a memoryless model), i.e., the transition to the next state is determined by the current state only and independent of previous states. The probability of occurrence for a call chain of length 3 or more can be computed as the product of transition probabilities of adjacent call pairs, because all the call transitions are assumed to be independent. Our approach can be generalized to higher-order Markov chains, which will improve the detection accuracy [71, 90, 26]. That generalization will require the estimation of higher order conditional probabilities and more complex computation.
- *Refined probability estimation* The probability estimation within control flow graph can be more accurate. Advanced program analysis techniques such as blended analysis [27, 28, 35] may be used to capture a more refined and accurate representation of the intended program behaviors. For the prediction of branch frequency, [10, 120] have shown that pure static program analysis can help predict the execution frequencies of branches and function calls, with heuristics identified from executable. These techniques can help improve the probability estimation operation and construct more accurate initial HMM models in STILO.

Bibliography

- [1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS '05, pages 340–353, New York, NY, USA, 2005. ACM.
- [2] Adobe Attack. <http://krebsonsecurity.com/2013/10/adobe-breach-impacted-at-least-38-million-users/>.
- [3] M. Atallah, R. Gwadera, and W. Szpankowski. Detection of significant sets of episodes in event sequences. In *Proceedings of the Fourth IEEE International Conference on Data Mining*, ICDM '04, pages 3–10, Washington, DC, USA, 2004. IEEE Computer Society.
- [4] Audit framework. https://wiki.archlinux.org/index.php/Audit_framework.
- [5] Cost of Security. http://institute.lanl.gov/isti/summer-school/cluster_network/projects-2011/2011%20Yellow%20TeamLopez%20Mortensen%20Chambers.pdf.
- [6] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 189–200, New York, NY, USA, 2008. ACM.
- [7] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 189–200, 2008.
- [8] G. K. Baah, A. Podgurski, and M. J. Harrold. Causal inference for statistical fault localization. In *International Symposium on Software Testing and Analysis*, pages 73–84, 2010.
- [9] A. Baliga, L. Iftode, and X. Chen. Automated containment of rootkits attacks. *Computers & Security*, 27(7-8):323–334, 2008.

- [10] T. Ball and J. R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 300–313, New York, NY, USA, 1993. ACM.
- [11] L. Baum. An inequality and associated maximization technique in statistical estimation for probabilistic functions of Markov processes. *Inequalities*, 3:1–8, 1972.
- [12] L. E. Baum, J. Eagon, et al. An inequality with applications to statistical estimation for probabilistic functions of markov processes and to a model for ecology. *Bull. Amer. Math. Soc*, 73(3):360–363, 1967.
- [13] L. E. Baum and T. Petrie. Statistical inference for probabilistic functions of finite state markov chains. *The annals of mathematical statistics*, pages 1554–1563, 1966.
- [14] L. E. Baum, T. Petrie, G. Soules, and N. Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of markov chains. *The annals of mathematical statistics*, pages 164–171, 1970.
- [15] L. E. Baum, G. R. Sell, et al. Growth transformations for functions on manifolds. *Pacific J. Math*, 27(2):211–227, 1968.
- [16] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 48–62, Washington, DC, USA, 2006. IEEE Computer Society.
- [17] R. P. L. Buse and W. Weimer. The road not taken: Estimating path execution frequency statically. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 144–154, Washington, DC, USA, 2009. IEEE Computer Society.
- [18] B. Calder, D. Grunwald, M. P. Jones, D. C. Lindsay, J. H. Martin, M. Mozer, and B. G. Zorn. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.*, 19(1):188–222, 1997.
- [19] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.
- [20] E. Chien. The new generation of targeted attacks, 2010. Keynote in Recent Advances in Intrusion Detection (RAID).
- [21] F. Cohen. Computer viruses: theory and experiments. *Comput. Secur.*, 6(1):22–35, Feb. 1987.
- [22] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of 19th International World Wide Web Conference*, 2010.

- [23] G. F. Cretu, A. Stavrou, M. E. Locasto, S. J. Stolfo, and A. D. Keromytis. Casting out demons: Sanitizing training data for anomaly sensors. In *IEEE Symposium on Security and Privacy*, pages 81–95. IEEE Computer Society, 2008.
- [24] M. Cruz. Most abused infection vector, 2008. Trend Micro. <http://blog.trendmicro.com/most-abused-infection-vector/>.
- [25] D. E. Denning. An intrusion-detection model. *IEEE Trans. Softw. Eng.*, 13(2):222–232, Feb. 1987.
- [26] J. A. du Preez. Efficient training of high-order hidden markov models using first-order representations. *Computer speech & language*, 12(1):23–39, 1998.
- [27] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 118–128, New York, NY, USA, 2007. ACM.
- [28] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive java applications. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 59–70, New York, NY, USA, 2008. ACM.
- [29] DYNINST binary instrumentation technology. <http://www.dyninst.org>.
- [30] M. Egele, E. Kirda, and C. Kruegel. Mitigating drive-by download attacks: Challenges and open problems. In *Proceedings of the Open Research Problems in Network Security (iNetSec)*, pages 52–62, 2009.
- [31] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the Sixth Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2009.
- [32] T. Fawcett and F. Provost. Activity monitoring: Noticing interesting changes in behavior. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 53–62, New York, NY, USA, 1999. ACM.
- [33] 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 *IEEE Symposium on Security and Privacy*, 2004.
- [34] 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*, SP '03, pages 62–, Washington, DC, USA, 2003. IEEE Computer Society.

- [35] M. Fisher, B. Dufour, S. Basu, and B. G. Ryder. Exploring the impact of context sensitivity on blended analysis. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.
- [36] Adobe Flash Player remote code execution. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-3456>.
- [37] S. Forrest, S. A. Hofmeyr, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.
- [38] 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*, SP '96, pages 120–, Washington, DC, USA, 1996. IEEE Computer Society.
- [39] J.-M. Francois. jahmm. <http://jahmm.googlecode.com/>, 2009.
- [40] D. A. Freedman. *Statistical Models and Causal Inference: A Dialogue with the Social Sciences*. Cambridge University Press, 2010.
- [41] V. Frías-Martínez, S. J. Stolfo, and A. D. Keromytis. Behavior-profile clustering for false alert reduction in anomaly detection sensors. In *ACSAC*, pages 367–376. IEEE Computer Society, 2008.
- [42] D. Gao, M. K. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 318–329, New York, NY, USA, 2004. ACM.
- [43] D. Gao, M. K. Reiter, and D. Song. Behavioral distance measurement using hidden markov models. In *Proceedings of the 9th International Conference on Recent Advances in Intrusion Detection*, RAID'06, pages 19–40, Berlin, Heidelberg, 2006. Springer-Verlag.
- [44] D. Gao, M. K. Reiter, and D. X. Song. Beyond output voting: Detecting compromised replicas using HMM-based behavioral distance. *IEEE Trans. Dependable Sec. Comput.*, 6(2):96–110, 2009.
- [45] S. Gianvecchio, M. Xie, Z. Wu, and H. Wang. Measurement and classification of humans and bots in internet chat. In *Proceedings of USENIX Security Symposium*, 2008.
- [46] J. T. Giffin, S. Jha, and B. P. Miller. Detecting manipulated remote call streams. In *Proceedings of the 11th USENIX Security Symposium*, pages 61–79, Berkeley, CA, USA, 2002. USENIX Association.
- [47] J. T. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Network and Distributed System Security Symposium (NDSS)*, 2004.

- [48] J. T. Giffin, S. Jha, and B. P. Miller. Automated discovery of mimicry attacks. In *Proceedings of the 9th International Conference on Recent Advances in Intrusion Detection*, RAID'06, pages 41–60, Berlin, Heidelberg, 2006. Springer-Verlag.
- [49] R. Gopalakrishna, E. H. Spafford, and J. Vitek. Efficient intrusion detection using automaton inlining. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP '05, pages 18–31, Washington, DC, USA, 2005. IEEE Computer Society.
- [50] C. Grier, S. Tang, and S. T. King. Secure web browsing with the OP web browser. In *IEEE Symposium on Security and Privacy*, May 2008.
- [51] R. Gummadi, H. Balakrishnan, P. Maniatis, and S. Ratnasamy. Not-a-Bot: Improving service availability in the face of botnet attacks. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NDSI)*, 2009.
- [52] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
- [53] HTML Object Memory Corruption Vulnerability. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-0249>.
- [54] Vulnerability Note VU#493881. US-CERT. <http://www.kb.cert.org/vuls/id/493881>.
- [55] T. Jaeger, J. Liedtke, and N. Islam. Operating system protection for fine-grained programs. In *Proceedings of the 7th USENIX Security Symposium proceedings*, 1998.
- [56] T. Jaeger, A. D. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proceedings of the 6th USENIX Security Symposium*, July 1996.
- [57] A. Jones and Y. Lin. Application intrusion detection using language library calls. In *Proceedings of the 17th Annual Computer Security Applications Conference, ACSAC '01*, pages 442–, Washington, DC, USA, 2001. IEEE Computer Society.
- [58] G. S. Kc and A. D. Keromytis. e-NeXSh: Achieving an effectively non-executable stack and heap via system-call policing. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2005.
- [59] W. Khreich, E. Granger, A. Miri, and R. Sabourin. A survey of techniques for incremental learning of HMM parameters. *Inf. Sci.*, 197:105–130, Aug. 2012.
- [60] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th Conference on USENIX Security Symposium, SSYM'09*, pages 351–366, Berkeley, CA, USA, 2009. USENIX Association.

- [61] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 11–11, Berkeley, CA, USA, 2005. USENIX Association.
- [62] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *In Proc. of the 8th European Symposium on Research in Computer Security*, pages 326–343. Springer-Verlag, 2003.
- [63] W. Lee, S. Stolfo, and K. Mok. A data mining framework for building intrusion detection models. In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, pages 120–132, 1999.
- [64] Z. Liang, W. Sun, V. N. Venkatakrishnan, and R. Sekar. Alcatraz: An isolated environment for experimenting with untrusted software. *ACM Trans. Inf. Syst. Secur.*, 12:14:1–14:37, January 2009.
- [65] P. Likarish, E. E. Jung, and I. Jo. Obfuscated malicious javascript detection using classification techniques. In *Proceedings of 4th International Conference on Malicious and Unwanted Software*, 2009.
- [66] Z. Liu, S. M. Bridges, and R. B. Vaughn. Combining static analysis and dynamic learning to build accurate intrusion detection models. In *Proceedings of the Third IEEE International Workshop on Information Assurance*, IWIA '05, pages 164–177, Washington, DC, USA, 2005. IEEE Computer Society.
- [67] M. T. Louw, J. S. Lim, and V. N. Venkatakrishnan. Enhancing web browser security against malware extensions. *Journal in Computer Virology*, 4(3):179–195, 2008.
- [68] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. Blade: An attack-agnostic approach for preventing drive-by malware infections. In *Proceedings of 17th ACM Conference on Computer and Communications Security*, 2010.
- [69] www.malwaredomainlist.com.
- [70] www.malwareurl.com.
- [71] J.-F. Mari, D. Fohr, and J.-C. Junqua. A second-order hmm for high performance word and phoneme-based continuous speech recognition. In *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, volume 1, pages 435–438. IEEE, 1996.
- [72] J. A. Morales, E. J. Kartaltepe, S. Xu, and R. S. Sandhu. Symptoms-based detection of bot processes. In I. V. Kottenko and V. A. Skormin, editors, *MMM-ACNS*, volume 6258 of *Lecture Notes in Computer Science*, pages 229–241. Springer, 2010.

- [73] A. Moshchuk, T. Bragin, and D. Deville. SpyProxy: Execution-based detection of malicious web content. In *Proceedings of the 16th USENIX Security Symposium*, 2007.
- [74] Microsoft high-risk extensions. <http://support.microsoft.com/kb/883260>.
- [75] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, Feb. 2006.
- [76] Nasdaq Attack. <http://www.reuters.com/article/2013/07/18/net-us-nasdaq-cybercrime-website-idUSBRE96H1F520130718>.
- [77] https://developer.mozilla.org/en/XUL_Overlays.
- [78] C. Parampalli, R. Sekar, and R. Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, ASIACCS '08, pages 156–167, New York, NY, USA, 2008. ACM.
- [79] A. Prakash, H. Yin, and Z. Liang. Enforcing system-wide control flow integrity for exploit detection and diagnosis. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 311–322, New York, NY, USA, 2013. ACM.
- [80] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser analysis of web-based malware. In *HotBots'07: Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, Berkeley, CA, USA, 2007. USENIX Association.
- [81] Apple QuickTime 7.3 RTSP Response Exploit, CVE-2007-6166, <http://securityevaluators.com/content/case-studies/sl/>.
- [82] L. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.
- [83] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 31–39, New York, NY, USA, 2010. ACM.
- [84] K. Rieck, G. Schwenk, T. Limmer, T. Holz, and P. Laskov. Botzilla: detecting the "phoning home" of malicious software. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1978–1984, New York, NY, USA, 2010. ACM.
- [85] <http://www.exploit-db.com/wp-content/themes/exploit/docs/28553.pdf>.
- [86] <http://phrack.org/issues/58/4.html>.
- [87] <https://github.com/JonathanSalwan/ROPgadget>.

- [88] <https://github.com/Overcl0k/rp>.
- [89] rp++. <https://github.com/Overcl0k/rp>.
- [90] M. Seifert. *Extensions of hidden Markov models for the analysis of DNA microarray data*. PhD thesis, Halle (Saale), Martin-Luther-Universitat Halle-Wittenberg, Diss., 2010, 2010.
- [91] 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*, SP '01, Washington, DC, USA, 2001. IEEE Computer Society.
- [92] M. Sharif, K. Singh, J. Giffin, and W. Lee. Understanding precision in host based intrusion detection: Formal analysis and practical models. In *Proceedings of the 10th International Conference on Recent Advances in Intrusion Detection*, RAID'07, pages 21–41, Berlin, Heidelberg, 2007. Springer-Verlag.
- [93] J. Shirley and D. Evans. The user is not the enemy: Fighting malware by tracking user intentions. In *Proceedings of New Security Paradigms Workshop (NSPW)*, pages 22–25, September 2008.
- [94] Software-artifact Infrastructure Repository. <http://sir.unl.edu/portal/index.php>.
- [95] skape. Understanding windows shellcode. <http://www.nologin.org/Downloads/Papers/win32-shellcode.pdf>, 2003.
- [96] skape. Metasploit's meterpreter. <http://www.nologin.org/Downloads/Papers/meterpreter.pdf>, 2004.
- [97] skape and J. Turkulainen. Remote library injection. <http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf>, 2004.
- [98] C. Song, J. Zhuge, X. Han, and Z. Ye. Preventing drive-by download via inter-module communication monitoring. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.
- [99] Y. Song, A. D. Keromytis, and S. J. Stolfo. Spectrogram: A mixture-of-Markov-chains model for anomaly detection in web traffic. In *NDSS*. The Internet Society, 2009.
- [100] A. Sotirov. Heap Feng Shui in JavaScript. <http://www.phreedom.org/research/heap-feng-shui/>.
- [101] E. H. Spafford and G. Kim. The design and implementation of tripwire: A file system integrity checker. In *2nd ACM Conf. on Computer and Communication Security (CCS)*, 1994.

- [102] S. Sparks, S. Embleton, R. Cunningham, and C. Zou. Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 477–486, 2007.
- [103] A. Stamminger, C. Kruegel, G. Vigna, and E. Kirda. Automated spyware collection and analysis. In P. Samarati, M. Yung, F. Martinelli, and C. A. Ardagna, editors, *ISC*, volume 5735 of *Lecture Notes in Computer Science*, pages 202–217. Springer, 2009.
- [104] D. Stefan, C. Wu, D. Yao, and G. Xu. Cryptographic provenance verification for the integrity of keystrokes and outbound network traffic. In *Proceedings of the 8th International Conference on Applied Cryptography and Network Security (ACNS)*, June 2010.
- [105] D. Stefan and D. Yao. Keystroke-dynamics authentication against synthetic forgeries. In *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, November 2010.
- [106] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. A. Kemmerer, C. Kruegel, and G. Vigna. Your botnet is my botnet: analysis of a botnet takeover. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *ACM Conference on Computer and Communications Security*, pages 635–647. ACM, 2009.
- [107] W. Sun, R. Sekar, Z. Liang, and V. N. Venkatakrishnan. Expanding malware defense by securing software installations. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment, DIMVA '08*, pages 164–185, Berlin, Heidelberg, 2008. Springer-Verlag.
- [108] Superbuddy exploits through AOL activeX control. <http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5820>.
- [109] K. M. C. Tan and R. A. Maxion. “Why 6?” defining the operational limits of Stide, an anomaly-based intrusion detector. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, SP '02, pages 188–, Washington, DC, USA, 2002. IEEE Computer Society.
- [110] Target Attack. <http://krebsonsecurity.com/tag/target-data-breach/>.
- [111] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 189–206, New York, NY, USA, 2011. ACM.
- [112] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, SP '01, pages 156–, Washington, DC, USA, 2001. IEEE Computer Society.

- [113] 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, CCS '02*, pages 255–264, New York, NY, USA, 2002. ACM.
- [114] H. J. Wang, X. Fan, J. Howell, and C. Jackson. Protection and communication abstractions for web browsers in MashupOS. In *ACM Symposium on Operating Systems Principle (SOSP)*, pages 1–16. ACM Press, 2007.
- [115] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 18th Usenix Security Symposium*, August 2009.
- [116] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2006.
- [117] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.
- [118] W.-K. Wong, A. Moore, G. Cooper, and M. Wagner. Rule-based anomaly pattern detection for detecting disease outbreaks. In *Eighteenth National Conference on Artificial Intelligence*, pages 217–223, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.
- [119] C. Wressnegger, G. Schwenk, D. Arp, and K. Rieck. A close look on N-grams in intrusion detection: Anomaly detection vs. classification. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, AISec '13*, pages 67–76, New York, NY, USA, 2013. ACM.
- [120] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture, MICRO 27*, pages 1–11, New York, NY, USA, 1994. ACM.
- [121] X86-64-instructions. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html?iid=tech_vt_tech+64-32_manuals.
- [122] H. Xiong, P. Malhotra, D. Stefan, C. Wu, and D. Yao. User-assisted host-based detection of outbound malware traffic. In *Proceedings of International Conference on Information and Communications Security (ICICS)*, December 2009.
- [123] K. Xu, D. Yao, Q. Ma, and A. Crowell. Detecting infection onset with behavior-based policies. In *Network and System Security (NSS), 2011 5th International Conference on*, pages 57–64. IEEE, 2011.

- [124] D.-Y. Yeung and Y. Ding. Host-based intrusion detection using dynamic and static behavioral models. *Pattern Recognition*, 36(1):229 – 243, 2003.
- [125] S. Zarandioon, D. Yao, and V. Ganapathy. OMOS: A framework for secure communication in mashup applications. In *ACSAC'08: Proceedings of the 24th Annual Computer Security Applications Conference*, pages 355–364, December 2008.
- [126] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 29–40. ACM, 2011.
- [127] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy*, pages 559–573, 2013.
- [128] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *Proceedings of the 22-nd USENIX Conference on Security, SEC'13*, pages 337–352, Berkeley, CA, USA, 2013. USENIX Association.