Probabilistic Program Modeling for High-Precision Anomaly Classification

Kui Xu   Danfeng (Daphne) Yao   Barbara G. Ryder   Ke Tian
Computer Science Department
Virginia Tech
Blacksburg, VA, 24060
Email: {xmenxk, danfeng, ryder, ketian}@vt.edu

Abstract—**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. We introduce a new anomaly-based detection technique that probabilistically models and learns a program's control flows for high-precision behavioral reasoning and monitoring. Our prototype in Linux is named *STILO*, which stands for STatically InitiaLized markOv. Experimental evaluation involves real-world code-reuse exploits and over 4,000 testcases from server and utility programs. STILO achieves up to 28-fold of improvement in detection accuracy over the state-of-the-art HMM-based anomaly detection. Our findings suggest that the probabilistic modeling of program dependences provides a significant source of behavior information for building high-precision models for real-time system monitoring.**

Keywords*—*Anomaly detection, static program analysis, hidden Markov model, probability

I. Introduction

New generations of code-reuse based hijacking techniques allow attackers to compose malicious control flows from victim 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.

The increasing sophistication in modern exploits demands precise program behavior modeling and runtime classification. In the paradigm of anomaly detection, one builds models to capture the expected execution patterns of programs. Program behaviors that deviate from the model indicate possible intrusions. Anomalies may be due to control-flow hijacking, unexpected inputs, or operational errors.

Program behavior models can be learned from execution traces. For example, one approach 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 [1], [2], [3] and library calls [4]. Any sequence with new calls or out-of-the-order calls is classified as an anomaly. However, a widely known limitation of n-gram is that the approach needs to enumerate and store all possible call sequences, which hurts its scalability.

There exist several scalable learning techniques for building program behavior models, for example the automaton model [5], [6], the hidden Markov model (HMM) [7], [8], [9], [10], and the execution-graph model [11]. 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 supports anomaly detection (i.e., whether a call sequence is feasible to occur or not), as well as quantification (e.g., how likely a sequence occurs in the normal program execution).

However, program behavior models constructed solely through learning from program traces (e.g., [6], [8], [12]) 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 [13], [14]. 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 ([15], [16]) are *complete*, in that all the statically feasible paths can be predicted.

Yet, 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 causes important signs of run-time program misuses or undesirable program-behavior changes to be ignored.

We set two goals for designing our program behavior model for anomaly-based detection:

- To use probabilistic reasoning to ascertain the likelihoods of occurrences.
- To cover both static and dynamic control-flow behaviors.

We present a new classification technique for detecting

This work has been supported by ONR grant N00014-13-1-0016.

anomalous program executions and call sequences. The classification is based on our new probabilistic control-flow model representing the expected call sequences of the program. The construction of this probabilistic control-flow model incorporates both statically and dynamically extracted control-flow information, resulting in nearly 30-fold improvement in anomaly detection accuracy in our experiments.

Our key enabler is the efficient and compact composition of the static program analysis results into an initialization matrix for the hidden Markov model. We design a new 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). These probabilities are used to customize our classification model, namely hidden Markov model. Our experiments show that this static customization significantly boosts the quality and coverage of the learner. Our detection system does not require any binary transformation of the program.

In comparison to the existing probabilistic program-modeling research (e.g., [17]), our main difference is that our probabilistic program analysis is driven by the goal of anomaly detection. Thus, our analysis is coupled with HMM-based classification. The contributions in this paper are summarized as follows.

1) We present a technique that can statically infer the probabilities associated with programs, specifically the transitions between calls (system or library). We give the first demonstration that these probability values are useful for guiding dynamic learning techniques towards more optimal configurations, significantly improving security guarantees. With our technique, learning models, such as HMM, are more resilient to the incompleteness of training traces.
2) 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**nitia**L**ized mark**O**v. We extensively compare the classification accuracy and performance of STILO with regular HMM models. Our evaluation is performed on over 4,000 test cases from eight Linux applications including a collection of utility programs, server programs `proftpd` and `nginx`.
3) STILO consistently outperforms the regular HMM models in classification accuracy, achieving 11- to 28-fold improvement on average. 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 (including 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 program anomaly detection.

Our work gives a new method for constructing program behavior models for anomaly detection that significantly enhances the detection capabilities of learning-based methods. This new modeling technique provides more effective tools for cyber defenders in battling against modern stealthy exploits.

II. Overview of Our Approach

The attack model in this work is focused on invalid and abnormal control flow of a program, e.g., executing injected code through unsanitized arguments or buffer overflow vulnerabilities, bypassing security checks, exploiting race conditions. 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 attachments).

In this section, we first illustrate the new technical challenges associated with probabilistic modeling of program call sequences and point out the deficiencies in existing and alternative approaches. Then, we give an overview of our design.

A. Challenges in Probabilistic Program Behavior Modeling

Let a function or program have 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 [8]).
- (*Con*) Incomplete training set results in false alarms. As shown in Figure 1, system call sequences containing rare but statically feasible path P_2 may be misclassified as abnormal.

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 1, a highly unlikely call sequence $P_2P_2P_2$ (an indicator of possible exploits) cannot be detected.

Straightforward attempts to unify the learning and static models are also problematic. Consider a straightforward hybrid approach for building a program behavior model, where one may use two independent models – a program analysis model (e.g., [15]) and a quantitative learning model (e.g., [1]) – 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.

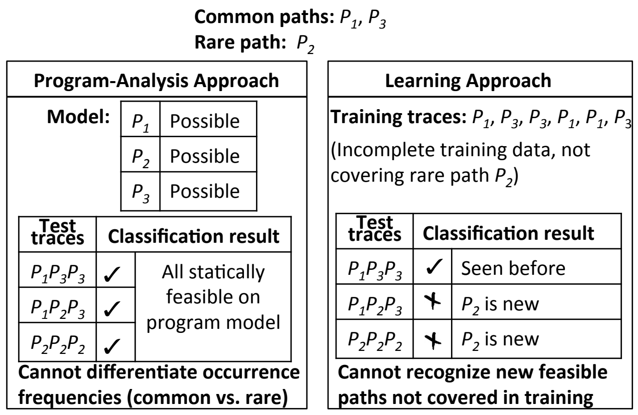
Common paths: P_1, P_3
Rare path: P_2
Program-Analysis Approach
Model:
P_1 Possible
P_2 Possible
P_3 Possible
Test traces
Classification result
$P_1P_3P_3$ ✓
$P_1P_2P_3$ ✓
$P_2P_2P_2$ ✓
All statically feasible on program model
Cannot differentiate occurrence frequencies (common vs. rare)
Learning Approach
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

Fig. 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.

Our method eliminates these deficiencies through a new program-analysis based probability forecast. **i)** With a probabilistic representation for call sequences, it differentiates their frequencies of occurrences, improving detection sensitivity. It computes a probability $P(\langle c_1, \ldots, c_k \rangle | \lambda)$ for an observed call sequence $\langle c_1, \ldots, c_k \rangle$ 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. **ii)** The new model has the potential to recognize legitimate new calls, as well as new call sequences that do not appear in the training set.

B. Key Steps of Our Algorithm

Our program-analysis-guided probabilistic detection has the capability to reason about the occurrence likelihoods beyond the binary feasibility prediction, useful for detecting and deterring stealthy attacks. A diagram illustrating our workflow is shown in Figure 2. We give an overview of our workflow below. Each step is described in details in the following sections.

1) **Probability Forecast:** We extract information from control-flow graphs to statically estimate likelihoods of occurrence for call sequences through two steps. 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. (Section III describes our probability forecast operation in details.)
   - *Step 1:* We take a control-flow graph of a function and outputs a *call-transition matrix* for this function (Definition 4). This matrix consists of estimated call transition probabilities, which represent the likelihoods of occurrence for sequences of calls when the function $f()$ is executed. Computing call-transition matrix is described in Section III-C.
   - *Step 2:* To obtain the call transitions of the entire program, we aggregate 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. Aggregating call transitions is described in Section III-D.
2) **Initialization:** 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. The values 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. Section IV describes this operation in details.
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.

Advanced mimicry attacks or attack sequences that are extremely short are challenging to detect. A hand-crafted mimicry attack was introduced in [18], 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. The advantage of static analysis is to provide a complete and quantitative initial representation of program behaviors, which is further trained with dynamic execution information to probabilistically characterize the control flow behaviors of a program.

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 [19], [20]. The latter are limited to pair-wise conditional probabilities on a control flow graph.

III. Probability Forecast of Call Sequences

In this section, we give formal probability definitions needed for analyzing control-flow graphs, and present algorithms for realizing control-flow probability forecast, specifically computing reachability probabilities and transition probabilities.

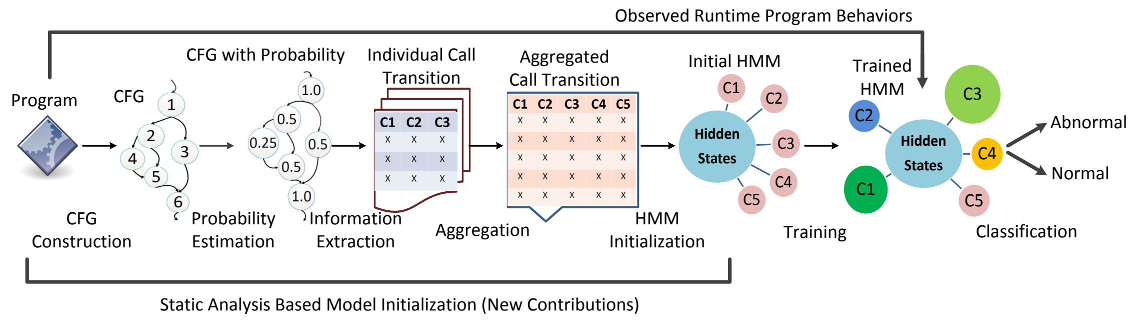

Fig. 2. A diagram illustrates our anomaly detection workflow.

Our static analysis' goal is to extract call transition properties to include in the program behavior model. Such a model is capable of recognizing new legitimate call sequences not seen during training, thus significantly improving the accuracy of detection.

A. Our Definitions

We give new probability definitions in the context of program execution. The definitions include the conditional probability of adjacent CFG nodes, the reachability probability from the function entry, and transition probability for a call pair. With these definitions, one can quantify control-flow properties in a rigorous representation that is compatible with Markov-chain based learning model.

Definition 1: The conditional probability P^c_{ij} 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 2: The reachability probability P^r_i 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.

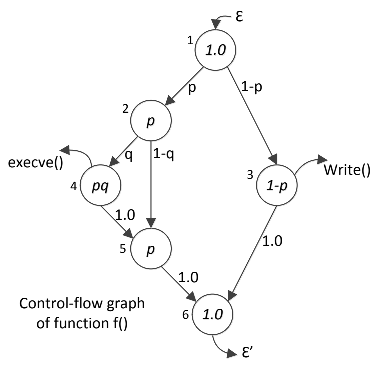

Fig. 3. Examples of conditional probabilities and reachability probabilities for function $f()$. Conditional probability of a node pair is shown on the edge. Reachability probability of a node is shown in the node. ϵ and ϵ' represent the external call site and return site of $f()$.

Examples of conditional and reachability probabilities of a simplified control-flow graph are given in Figure 3.

Definition 3: The transition probability P^{tf}_{ij} 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.

To compute these values, our method first traverses the control-flow graph of a function to statically approximate the *conditional probability* P^c_{ij} for each pair of adjacent nodes $(n_1\ \rightarrow\ n_2)$. Then, based on conditional probabilities, our algorithm computes the *reachability probability* P^r_i 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. Details are given in the next few sections.

We define the *call-transition matrix of a function* in Definition 4. The call-transition probability is defined for a call pair (c_1, c_2), where c_1 precedes c_2.

Definition 4: 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^t_{ij}.

Table I shows an example of the call-transition matrix of the function in Figure 3.

TABLE I. A CALL-TRANSITION MATRIX OF THE FUNCTION IN FIGURE 3. ϵ REPRESENTS THE EXTERNAL CALLER OF THIS FUNCTION. ϵ' REPRESENTS THE EXTERNAL RETURN SITE.

	ϵ'	write	execve
ϵ	$p(1-q)$	$1-p$	pq
write	$1-p$	0	0
execve	pq	0	0

B. Computing Reachability Probability

Our computation traverses a CFG and estimates the probability to reach a CFG node from the function entry, conditioning on the function being executed with probability 1.0. The probabilities are normalized at the aggregation operation later.

The calculation of reachability probabilities is top down starting from the function entry of CFG. To compute the probability of a child node, one needs the reachability values of its parents. We perform the topological sorting on all nodes and our reachability-probability computation follows the topological order.

Formally, for node n_k the reachability probability P^r_k is computed as in Equation 1, where P^r_i is the reachability probability of one of n_k's parents and P^c_{ik} is the conditional probability for node pair (n_i, n_k).

$$P^r_k = \sum_{\forall\ n_i\ \in\ \texttt{parent set of } \texttt{n}_\texttt{k}} P^r_i * P^c_{ik} \tag{1}$$

Specifically, P^c_{ij} 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^c_{ij} 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 [21], [22], [23], path frequency [24].

We illustrate the probability values for the control-flow graph for function $f()$ in Figure 3. P^r_5 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.

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.

C. Computing Call-Transition Matrix

We compute the likelihoods of occurrence for call pairs in a function, i.e., *transition probability*, based on reachability probabilities.

To compute the transition probability $P^{t_f}_{ab}$ 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}, ... , 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 $P^{t_f}_{a_k b_l}$ of call pair (c_a, c_b) in $f()$ as Equation (2).

$$P^{t_f}_{a_k b_l} = P^r_k * \prod\nolimits_{i=k}^{l-1} P^c_{i(i+1)} \tag{2}$$

In a context-sensitive model as shown in Equation (2), the calling context is recorded when computing a call-transition probability. In other words, two calls made at different call sites are considered different, even when the calls are the same. 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 (3). Our STILO prototype is flow-sensitive and context-insensitive. Thus, the aggregation follows Equation (3). Enhancing the sensitivity of calling context is our ongoing work.

$$P^{t_f}_{ab} = \sum_{\substack{\forall\ \texttt{node pairs } (\texttt{n}_\texttt{k},\texttt{n}_\texttt{l}) \\ \texttt{s.t. } \texttt{n}_\texttt{k} \texttt{ calls } \texttt{c}_\texttt{a},\texttt{n}_\texttt{l} \texttt{ calls } \texttt{c}_\texttt{b}}} P^{t_f}_{a_k b_l} \tag{3}$$

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

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

Definition 5: Properties of call-transition matrix of a function:

1) The sum of the first row of a call-transition matrix of function $f()$ must sum to 1, i.e., $\sum_i P^{t_f}_{\epsilon i} = 1$. Similarly, the sum of the first column of a call-transition matrix of function $f()$ must sum to 1, i.e., $\sum_j P^{t_f}_{j\epsilon'} = 1$. This property is because $f()$ 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 must equal the sum of its outgoing probabilities, i.e., $\sum_j P^{t_f}_{ji} = \sum_k P^{t_f}_{ik}$.

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).

D. Aggregation of Call Transitions

The final step in our probability forecast 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 used to initialize the Markov-based learning model.) Aggregation operation takes as inputs **i)** the call graph of the program and **ii)** call-transition matrix for each function. The call graph is needed for the calling relations among functions.

1) Tasks and Complexity: Statically constructed flow-sensitive automata may have formidable complexity, if one needs to capture all the statically feasible paths in a program. The total number of states in the automata grows quickly with the size of its corresponding program, and the possible execution paths are exponential. E.g., $O(m^k)$ number of different nodes and paths are available for a program's automata with average execution path of length k and average out-degree of m for each node. [1]

Our matrix on call transition properties is extremely compact. For space complexity, the dimension (row or column) of our aggregated matrix is the number of distinct calls. The matrix records pair-wise call transitions, as opposed to the entire call sequences. All occurrences of the same call pair are aggregated to one matrix cell value. The space complexity is $O(n^2)$, where n is the number of distinct calls from the static analysis of a program. Our aggregation operation has two tasks:

1. 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

[1]To reduce the space overhead, the IAM model [15] performs heuristic automata compaction techniques such as merging similar states and reducing irrelevant states.

the rows and columns of the call-transition matrix. The call relations are obtained from the call graph of the program.

2. 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.

The aggregated call-transition matrix should also satisfy the rules of probability in Definition 5.

2) Aggregation Algorithm: We distinguish three cases of call pairs during the probability aggregation, as illustrated in Figure 4. 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 $f()$.

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. For the space complexity, the dimension (row or column) of the compact aggregated matrix is the number of distinct calls. The matrix records pair-wise call transitions, as opposed to the entire call sequences. It is much more efficient than inlining control flow graphs [15], because all occurrences of the same call pair are added together to one matrix cell.

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 $(f_1(), f_2(), \ldots, f_n())$ 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 $i = n, \ldots, 2$. Pseudocode for aggregating call-transition matrices is in Algorithm 1. We prove that the matrix produced by our algorithm satisfies the probability rules in Definition 5 in the appendix.

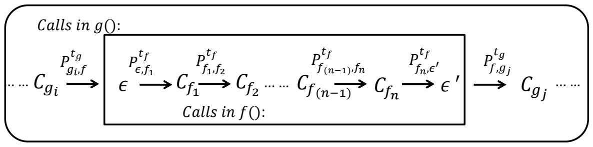

Fig. 4. Illustration of call sequences in a caller function g() and a callee function f(). Indices are from topological sort. The AGGREGATION operation replaces and expands entries with f() in g()'s call-transition matrix with calls in f().

E. Detailed Explanation 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 $P^{t_g}_{g_i,f_k} = P^{t_g}_{g_i,f} * P^{t_f}_{\epsilon,f_k}$, where $P^{t_f}_{\epsilon,f_k}$ 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 $P^{t_g}_{g_i,f_k} + P^{t_g}_{g_i,f} * P^{t_f}_{\epsilon,f_k}$, where $P^{t_g}_{g_i,f_k}$ 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 $P^{t_g}_{f,g_j} * P^{t_f}_{f_l,\epsilon'}$, where $P^{t_f}_{f_l,\epsilon'}$ 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 $P^{t_g}_{f_l,g_j} + P^{t_g}_{f,g_j} * P^{t_f}_{f_l,\epsilon'}$, where $P^{t_g}_{f_l,g_j}$ 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 $P^{t_f}_{f_k,f_l}$ 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 columns and rows for c_{f_k} and c_{f_l} and compute the new transition probability of (c_{f_k}, c_{f_l}) as $P^{t_g}_{f_k,f_l} = (\sum_i P^{t_g}_{g_i,f}) * P^{t_f}_{f_k,f_l}$. **ii)** Otherwise, compute the new transition probability for (c_{f_k}, c_{f_l}) as $P^{t_g}_{f_k,f_l} + (\sum_i P^{t_g}_{g_i,f}) * P^{t_f}_{f_k,f_l}$, where $P^{t_g}_{f_k,f_l}$ is the probability in $g()$'s matrix before the aggregation.

Algorithm 1 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
```

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: $P^{t_g}_{g_i,g_j} + P^{t_g}_{g_i,f} * P^{t_f}_{\epsilon,\epsilon'} * P^{t_g}_{f,g_j}$, where $P^{t_g}_{g_i,g_j}$ 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 $f()$. The two properties (Definition 5) of call-transition matrix are preserved during aggregation, which we show in the appendix.

Summary Our probability forecast takes as inputs control flows that are statically inferred, and transforms them into a rigorous probability representation. This static representation quantitatively characterizes the behaviors of a program and is in a format that can be naturally integrated into the corresponding HMM-based detection model. Loop analysis is not included, as we traverse each node once. Program behaviors that are not covered by our static program analysis (e.g., function pointer, recursions and loops) are learned from program traces by our STILO HMM model. We describe how STILO HMM utilizes the obtained probability values next.

IV. HMM Initialization

A limitation in existing HMM-based anomaly detection models (e.g., [8], [9]) is its reliance on training traces. The program behavioral model is constructed solely based on traces. The model's initial probabilities are chosen randomly – hoping they are corrected during training. As we demonstrate through experiments, the model's accuracy suffers from this simple initialization.

Our technique eliminates this deficiency. Our hidden Markov model encompasses **both static and dynamic prediction** of the program's behaviors. It is strategically initialized with the call-transition probabilities and call information obtained from the static program analysis. This approach significantly enhances the model's ability to discern execution anomalies, validating our hypothesis.

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., [8], [9]), N is the approximated number of distinct calls in program traces (which is usually smaller [2]).

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 $\{P^t_{ij}\}$ 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 ($\sum_j P^t_{ij}$) and normalized.

Impact of threshold selection on security. The choice of probability threshold T used to discern abnormal from normal segments has direct impact on security. Only segments having production probabilities greater than threshold T are classified as normal. In our experiments, we show how threshold values impact false-positive and false-negative rates. For example, smaller thresholds likely produce fewer false positives (i.e., false alarms), but may generate more false negatives (i.e., missed detection). In contrast, larger thresholds have the opposite impact on security, i.e., more false positives and fewer false negatives. Attackers may be able to evade the detection, if they can find exploit sequences whose probabilities are above the threshold, assuming that all the detection algorithms and parameters are public. This property is unavoidable, because of the intrinsic arms-race nature of security detection.

V. Experimental Evaluation

We name our prototype *STILO*, short for **ST**atically **I**nitia**L**ized mark**O**v. STILO is implemented in C/C++ using the `Dyninst` library [25]. Our experiments aim to answer the following questions.

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

A. Experimental 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) [13], as well as a FTP server `proftpd` and an HTTP server `nginx`. [3] For `proftpd` and `nginx`, we collected traces by manually interacting with the servers with a wide variety of file-transfer related tasks and web browsing tasks, respectively. The programs we tested include both utility applications and server programs, which are all potential victims of attacks

[2]Our experiments show a larger N does not guarantee the improvement in classification.

[3]These programs average over $52,586$ lines of code, and $1,139$ KB in size.

such as memory corruption, back-door, or binary instrumentation/replacement by attackers.

We compare the classification performance of STILO with the widely accepted HMM-based classification, which is the state-of-the-art probabilistic anomaly detection model (e.g., [8], [9]). We refer to that model as the regular HMM model.

The good coverage of test cases in SIR [4] 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.

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.

All standard HMM procedures are followed for model training and testing. We perform *10-fold cross validation* on 80% the normal traces. At each training iteration, convergence test is performed on the rest 20% of the normal traces. All comparable HMM models are subject to the same convergence criteria during training.

Given a threshold T for a program, false negative (FN) and false positive (FP) rates in HMM are defined in Equations (4) and (5), 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.

$$FN = \frac{|\{S_A : P_{S_A} > T\}|}{|\{S_A\}|} \tag{4}$$

$$FP = \frac{|\{S_N : P_{S_N} \leq T\}|}{|\{S_N\}|} \tag{5}$$

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

- *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 eight programs are evaluated.
- *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 evaluated. Not all of *Abnormal-A* segments contain exploits.

[4]Branch coverage is 67% on average and line coverage is 64% on average.

- *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.

We use the system tools `strace` and `ltrace` to intercept system calls and library calls of running application processes. [5] The HMM training and evaluation code is written in Java using the `Jahmm` library [28]. For identifying system calls, we compile programs with static linking. The library calls of interest are the `glibc` library calls. The call space contains over 200 distinct system calls and over 1,000 distinct library calls.

B. 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 server programs `proftpd` and `nginx` as an example in Figure 5. Experiments on the other six utility programs exhibit similar patterns. The details are shown in Figure 6.

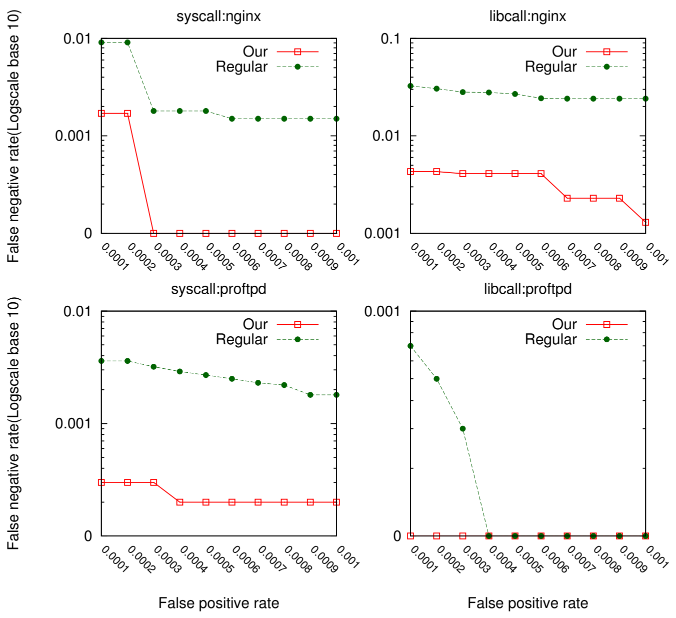

Fig. 5. Comparison of STILO and regular HMM's false negative rates (in Y-axis, base-10 log scale) for server programs *proftpd* and *nginx* on system calls and library calls under the same false positive rates (in X-axis).

[5]For performance consideration, alternative monitoring tools (e.g., auditd [26]) can be used by STILO in production systems. An acceptable 10% overhead was reported on a hybrid benchmark with realistic workload for auditd [27]. More performance discussion is in Section V-E.

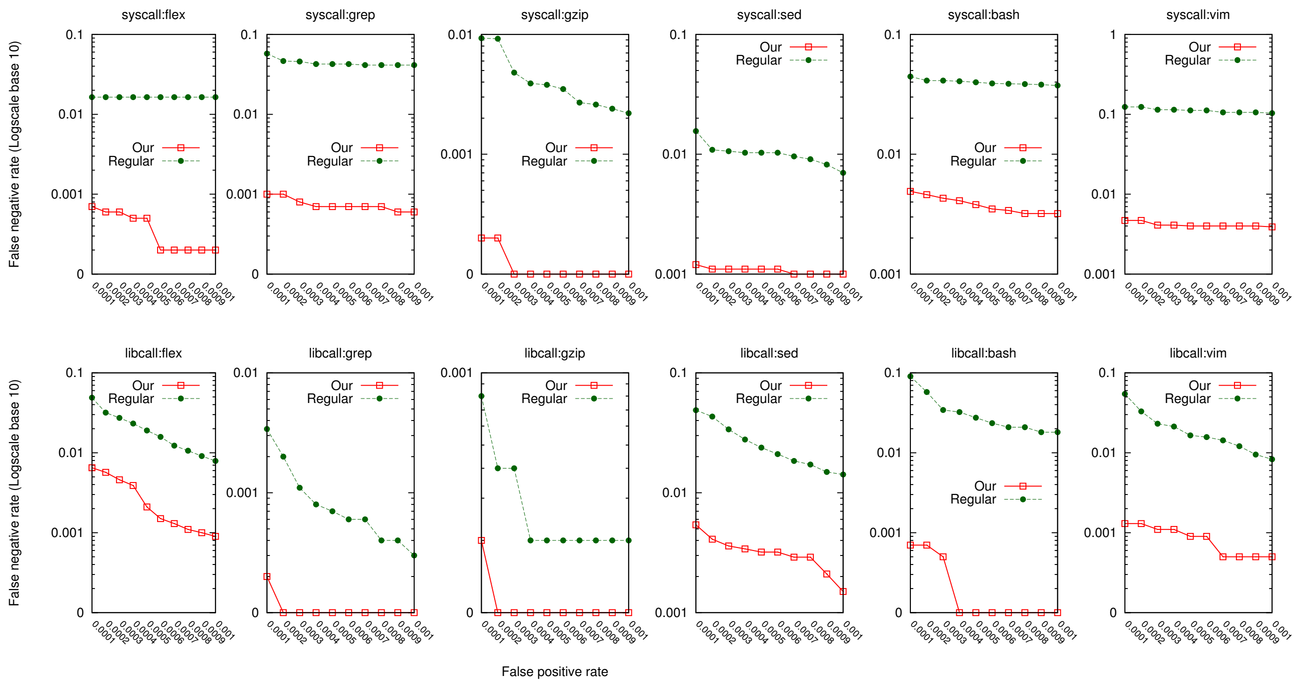

Fig. 6. Comparisons of classification accuracy in our model and the regular HMM for library and system calls. X-axis shows false positive rates (misclassified normal segments). Y-axis (logscale, base 10) shows false negative rates (misclassified abnormal segments).

Figure 7 shows the averaged false positive and false negative rates of STILO and the regular HMM for syscalls and libcalls. The average is computed across all eight programs evaluated. 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.

Our results show that 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.*

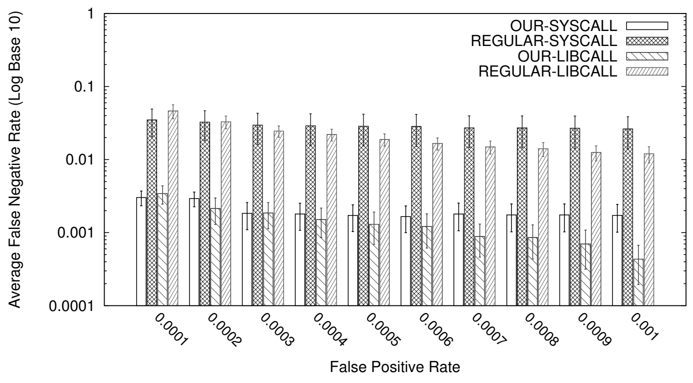

Fig. 7. Comparison of averaged false negative rates (in Y-axis) across eight programs evaluated on system calls or library calls by our model and the regular HMM, with respect to false positive rates (in X-axis). Standard errors are shown.

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 3.1 times more).

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.

C. Impact of Hidden States On Detection

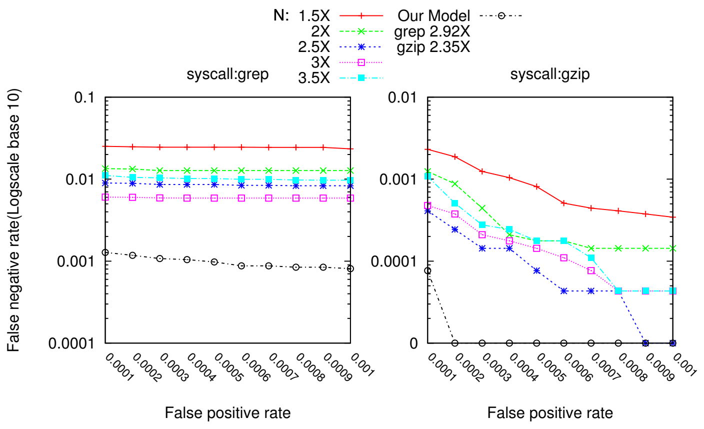

Fig. 8. The (unpredictable) impact of the number N of hidden states on classification accuracy. Classification results of system calls by regular HMM with various numbers of hidden states for grep and gzip programs are shown, as well as results from STILO HMM. For each HMM, N is shown as multiples (X) of the number of distinct calls in traces.

We evaluate the regular models with different numbers N of hidden states, and compare their classification accuracy with STILO for all programs at both syscall and libcall levels. The syscall results for `grep` and `gzip` are shown in Figure 8. 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.
- 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.

TABLE II. STILO SUCCESSFULLY DETECTS *Abnormal-A* SEGMENTS FROM REAL-WORLD EXPLOITS. IT RECOGNIZES ATTACK SEQUENCES WITHOUT TRIGGERING ANY FALSE ALERTS ON *Normal* SEGMENTS.

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

D. Detection of Code-Reuse Exploits

To demonstrate the ability to detect subtle code-reuse exploits, we reproduced `ROP` and `return_to_libc` exploits. 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 II.

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`. 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.

We list the classification probabilities by both our and regular models for such call segments. The false positive rate is set to 0.0001, and the corresponding thresholds for STILO and regular models are shown in Table III.

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. Another ROP exploit's payload executes shell commands attempting to steal sensitive information from the victim host. STILO detects all these attack call traces.

TABLE III. 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).

Segments	*Prob* (STILO)	*Prob* (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

Backdoor. For `proftpd` server, we reproduced a backdoor vulnerability (OSVDB-69562) and a buffer overflow (CVE-2010-4221) exploit. 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 evaluated are:

- [`read`, `read`, `close`, `unmap`, `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`]

E. Runtime Performance

Our static analysis for HMM initialization is efficient and takes seconds to finish. The runtime of STILO's STATIC CFG CONSTRUCTION, PROBABILITY ESTIMATION, and AGGREGATION OF CALL TRANSITION MATRIX operations is shown in Table IV.

The classification of a 15-call segment is fast (e.g., average 0.038 milliseconds for `gzip` on the system call model). The classification can also be made parallel with multithreaded programming for accelerated processing.

Training HMM models is generally time-consuming. For regular HMM, 10-fold cross-validation procedure may take several days to complete, e.g. for `proftpd`. We observe that STILO HMMs take fewer iterations to reach convergence than regular HMMs (39% fewer on average), reducing training time.

Intercepting calls with `strace` and `ltrace` introduce significant runtime overhead, which makes them infeasible for production systems in practice. Replacing them with other more sophisticated tools (e.g., `auditd` for system call tracing) will likely bring substantial reduction in runtime overhead.

TABLE IV. 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.

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

F. Summary of Experimental Findings

Our experimental findings positively confirm our hypothesis that control-flow information extracted from static program analysis can significantly improve 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 and server programs `proftpd` and `nginx`, for both library calls and system calls as shown (in Figure 7). 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 on average 39% few iterations to converge than regular HMM. This result shows that our initialization method facilitates the convergence during HMM training.
2) *STILO outperforms 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 8. 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) *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.
4) *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 7). 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.

VI. Related Work

Following the taxonomy in [15], 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 runtime). How models are constructed, through program analysis or learning, further differentiates them.

Learning-based or hybrid flow-sensitive models. Automaton-based models [5], [6] and HMM-based models [8], [9] are flow-sensitive anomaly detection models. With a sufficient large n, n-gram models (e.g., [2]) are also flow-sensitive. The execution-graph model in [11] was built through learning runtime program execution patterns (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 [5], where researchers refined the basic statically generated model with program traces in order to cover new transitions associated with runtime properties, such as exception handlers and dynamic libraries. In comparison, our technique is centered on probabilistic reasoning of program behaviors, whereas [5] is not a probabilistic approach, thus their automatons do not have the capability to record, model or analyze occurrence frequencies.

Probabilistic data mining techniques were demonstrated for analyzing network intrusions in [29]. The first probabilistic learning work for program behavior modeling was presented by Warrender *et al.* [8] using a hidden Markov model for classification system call segments, which we extensively compare with throughout the paper. Later, researchers proposed to use an HMM for comparing two parallel executions for anomaly detection [7].

Program analysis-based flow-sensitive models. Instead of learning the automaton model from program traces, 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 [16]) and later improved by others (e.g., inline automaton model (IAM) in [15]). Dyck model [30] described how flow-sensitive and context-insensitive NFA can enjoy context sensitivity (more below).

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 runtime overhead. For example, as shown in [16] building a context-sensitive push-down automaton (PDA) (in their *abstract stack* model) has prohibitive runtime costs. As pointed out by [15], context sensitivity does not imply flow sensitivity, and vice versa.

Using program counter [6] or call stack information (e.g., dynamically constructed in VtPath [31] or statically constructed and more precise in VPStatic [32]) to distinguish calling context have been shown efficient in practice. Several techniques for improving context sensitivity of NFA were proposed in [33], some of which require program instrumentation such as renaming system calls to distinguish different invocations of the same functions. Dyck model [30] inserted code that links the entry and exit of a target function with its call sites. This instrumentation differentiates call sites, improving context sensitivity.

Existing papers on context-sensitivity improvement presented fan-in properties, as opposed to runtime classification results. Our current STILO prototype is flow-sensitive, but context-insensitive. Integrating the above techniques into STILO to provide varying degrees of context sensitivity is feasible.

Integrity properties and enforcement. The property of control-flow integrity (CFI) generally refers to that program execution must follow a path of a pre-determined CFG (e.g., CFGs derived from static binary analysis) [34]. Enforcement of CFI property can be realized through modifying source and destination instructions associated with control-flow transfers and embedding control-flow policies with IDs within the binary for runtime enforcement [34]. Subsequent CFI techniques improve on the handling of forward edges (an indirect jump or call) in the control-flow graph [35] and the detection of kernel rootkits [36]. Researchers proposed to use static analysis to reduce CFI's overhead [37]. Zhang and Sekar presented static analysis based methods and instrumentation to enforce the CFI property on commercial off-the-shelf binaries [38]. Total-CFI is a framework for system-wide runtime control-flow integrity enforcement built on a software emulator [39]. Special duplication techniques on functions and function pointers were demonstrated for preventing control-flow hijacking [40].

In comparison to these CFI techniques, our monitoring system is focused on the call-making portion of control flow instead of all the execution transfer instructions. We do not require any binary transformation or software emulator. Most CFI implementations assume limited dynamic code behaviors (such as self-modifying code, runtime code generation and loading). 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. Recent research showed possible gadget formation under CFI verification [41], confirming the need for complementary runtime monitoring techniques such as ours.

Write integrity testing (WIT) technique aims to prevent memory-error exploits [42]. It predicts writable objects through static point-to analysis. WIT also realizes control-flow integrity and ensures that runtime indirect control transfers are consistent with control-flow graphs.

Data flow. Our work is focused on system-call specific control flows. In the literature, data flows together with control flows were shown useful for anomaly detection [43]. Def-use data-dependence analysis has been used for modeling malware behaviors, e.g., [44], [45], [46]. Researchers demonstrated the effectiveness of modeling arguments of system calls for anomaly detection, e.g., in terms of the distribution of string lengths and characters [47].

The data-flow integrity (DFI) property, first proposed by Castro, Costa, and Harris, refers to the consistency requirement between runtime data flow and statically predicted data flow [48]. The authors demonstrated the detection of both control and non-control-data attacks by DFI enforcement.

Other probabilistic approaches. Probabilistic programming is designed for providing automatic inference on user-specified probabilistic models [49]. Associated techniques were proposed for inferring properties of probabilistic programs [50]. Researchers have also used probabilistic programming languages to analyze information leakage [51], [52]. Our current STILO model does not handle probabilistic programs. How to extend it to protect probabilistic programs is an interesting open question.

Probabilistic abstract interpretation has been used to compute and limit the knowledge gain associated with information release [53]. The work by Sankaranarayanan, Chakarov, and Gulwani statically approximated probabilities of program-path execution with Monte-Carlo simulation [17]. Sampson *et al.* provided a framework for expressing and verifying probabilistic assertions of variables in programs with a Bayesian-network based model [54]. Recently, probabilistic modeling was proposed to predict program properties in new, unseen programs (aka Big Code) [55]. Big Code is not specifically designed for control-flow security. Thus, it is unclear how it can be extended for program anomaly detection.

VII. Conclusions, Future Work, and An Open Problem

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. We provided 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.

For future work, we plan to explore the use of probabilistic automaton (e.g., [56]) in the detection, specifically constructing program behavioral models with static-program-analysis enhanced probabilistic automata. We also plan to support the incremental learning [57] in STILO to achieve adaptive detection.

An open problem The custom built HMM in the behavioral distance measurement work by Gao, Reiter, and Song [7] has *pairs* of systems call segments as observed symbols (as opposed to single system call segments). The model measures the behavioral distance between two program variants (e.g., Linux web server and Windows web server). This approach is generally known as *N-variant* [58]. Their HMM is initialized with random probability distributions and a fixed number of hidden states. Then, the initialized model is trained with benign traces (in the form of pairs of system call segments). How to extend STILO-HMM to the N-variant context is an interesting open problem.

Acknowledgment

The authors would like to thank David Evans and anonymous reviewers for their insightful comments and suggestions on the work.

References

[1] 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*, ser. SP '96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 120–.

[2] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, no. 3, pp. 151–180, 1998.

[3] 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*, ser. AISec '13. New York, NY, USA: ACM, 2013, pp. 67–76. [Online]. Available: http://doi.acm.org/10.1145/2517312.2517316

[4] A. Jones and Y. Lin, "Application intrusion detection using language library calls," in *Proceedings of the 17th Annual Computer Security Applications Conference*, ser. ACSAC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 442–. [Online]. Available: http://dl.acm.org/citation.cfm?id=872016.872148

[5] 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*, ser. IWIA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 164–177. [Online]. Available: http://dx.doi.org/10.1109/IWIA.2005.6

[6] 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*, ser. SP '01. Washington, DC, USA: IEEE Computer Society, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=882495.884433

[7] 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.*, vol. 6, no. 2, pp. 96–110, 2009.

[8] C. Warrender, S. Forrest, and B. A. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *IEEE Symposium on Security and Privacy*, 1999, pp. 133–145.

[9] D.-Y. Yeung and Y. Ding, "Host-based intrusion detection using dynamic and static behavioral models," *Pattern Recognition*, vol. 36, no. 1, pp. 229 – 243, 2003. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0031320302000262

[10] Y. Dou, K. Zeng, Y. Yang, and D. Yao, "MadeCR: Correlation-based malware detection for cognitive radio," in *Proceedings of IEEE Conference on Computer Communications (INFOCOM)*, April 2015.

[11] 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*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 318–329. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030126

[12] H. Zhang, D. D. Yao, and N. Ramakrishnan, "Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery," in *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, S. Moriai, T. Jaeger, and K. Sakurai, Eds. ACM, 2014, pp. 39–50. [Online]. Available: http://doi.acm.org/10.1145/2590296.2590309

[13] Software-artifact Infrastructure Repository. http://sir.unl.edu/portal/index.php.

[14] 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*, ser. OOPSLA '11. New York, NY, USA: ACM, 2011, pp. 189–206. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048083

[15] 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*, ser. SP '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 18–31. [Online]. Available: http://dx.doi.org/10.1109/SP.2005.1

[16] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, ser. SP '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 156–. [Online]. Available: http://dl.acm.org/citation.cfm?id=882495.884434

[17] S. Sankaranarayanan, A. Chakarov, and S. Gulwani, "Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 447–458. [Online]. Available: http://doi.acm.org/10.1145/2491956.2462179

[18] 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*, ser. CCS '02. New York, NY, USA: ACM, 2002, pp. 255–264. [Online]. Available: http://doi.acm.org/10.1145/586110.586145

[19] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *International Symposium on Software Testing and Analysis*, 2010, pp. 73–84.

[20] 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*, 2007, pp. 477–486. [Online]. Available: http://www.acsa-admin.org/2007/papers/22.pdf

[21] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, ser. PLDI '93. New York, NY, USA: ACM, 1993, pp. 300–313. [Online]. Available: http://doi.acm.org/10.1145/155090.155119

[22] 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.*, vol. 19, no. 1, pp. 188–222, 1997.

[23] Y. Wu and J. R. Larus, "Static branch frequency and program profile analysis," in *Proceedings of the 27th annual international symposium on Microarchitecture*, ser. MICRO 27. New York, NY, USA: ACM, 1994, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/192724.192725

[24] 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*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 144–154. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070516

[25] DYNINST binary instrumentation technology. http://www.dyninst.org.

[26] Audit framework. https://wiki.archlinux.org/index.php/Audit_framework.

[27] M. Chambers, K. Lopez, and C. Mortensen, cost of Security (Auditing Focus). http://institute.lanl.gov/isti/summer-school/cluster_network/projects-2011/2011YellowTeam_LopezMortensenChambers.pdf.

[28] J.-M. Francois, "jahmm," http://jahmm.googlecode.com/, 2009.

[29] 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*, 1999, pp. 120–132.

[30] J. T. Giffin, S. Jha, and B. P. Miller, "Efficient context-sensitive intrusion detection," in *Network and Distributed System Security Symposium (NDSS)*, 2004.

[31] 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*, ser. SP '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 62–. [Online]. Available: http://dl.acm.org/citation.cfm?id=829515.830554

[32] 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.

[33] J. T. Giffin, S. Jha, and B. P. Miller, "Detecting manipulated remote call streams," in *Proceedings of the 11th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2002, pp. 61–79. [Online]. Available: http://dl.acm.org/citation.cfm?id=647253.720282

[34] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity: Principles, implementations, and applications," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, ser. CCS '05. New York, NY, USA: ACM, 2005, pp. 340–353. [Online]. Available: http://doi.acm.org/10.1145/1102120.1102165

[35] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 941–955. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice

[36] N. L. Petroni and M. W. Hicks, "Automated detection of persistent kernel control-flow attacks," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2007, p. 103115.

[37] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *ACM Conference on Computer and Communications Security*, Y. Chen, G. Danezis, and V. Shmatikov, Eds. ACM, 2011, pp. 29–40.

[38] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the 22-nd USENIX Conference on Security*, ser. SEC'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 337–352. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534766.2534796

[39] 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*, ser. ASIA CCS '13. New York, NY, USA: ACM, 2013, pp. 311–322. [Online]. Available: http://doi.acm.org/10.1145/2484313.2484352

[40] J. Noorman, N. Nikiforakis, and F. Piessens, "There is safety in numbers: Preventing control-flow hijacking by duplication," in *Secure IT Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7617, pp. 105–120. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-34210-3_8

[41] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 575–589. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.43

[42] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 263–277. [Online]. Available: http://dx.doi.org/10.1109/SP.2008.30

[43] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow anomaly detection," in *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, ser. SP '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 48–62. [Online]. Available: http://dx.doi.org/10.1109/SP.2006.12

[44] K. O. Elish, D. Yao, B. G. Ryder, and X. Jiang, "Profiling user-trigger dependence for Android malware detection," *Computers & Security*, vol. 49, pp. 255–273, March 2015.

[45] K. Elish, D. Yao, and B. G. Ryder, "On the need of precise inter-app ICC classification for detecting Android malware collusions," in *Proceedings of IEEE Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy*, May 2015.

[46] 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*. USENIX Association, 2009, pp. 351–366. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855768.1855790

[47] 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*. Springer-Verlag, 2003, pp. 326–343.

[48] M. Castro, M. Costa, and T. Harris, "Securing software by enforcing data-flow integrity," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 147–160. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298470

[49] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *Proceedings of the Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, J. D. Herbsleb and M. B. Dwyer, Eds. ACM, 2014, pp. 167–181. [Online]. Available: http://doi.acm.org/10.1145/2593882.2593900

[50] G. Claret, S. K. Rajamani, A. V. Nori, A. D. Gordon, and J. Borgstroem, "Bayesian inference for probabilistic programs via symbolic execution," Microsoft Research, Tech. Rep. MSR-TR-2012-86, 2012.

[51] P. Mardziel, M. S. Alvim, and M. Hicks, "Adversary gain vs. defender loss in quantified information flow," in *Proceedings of the International Workshop on Foundations of Computer Security (FCS)*, Jul. 2014.

[52] P. Mardziel, M. S. Alvim, M. Hicks, and M. Clarkson, "Quantifying information flow for dynamic secrets," in *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2014.

[53] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa, "Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation," *J. Comput. Secur.*, vol. 21, no. 4, pp. 463–532, Jul. 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2590624.2590625

[54] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 112–122. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594294

[55] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "Big Code"," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 111–124. [Online]. Available: http://doi.acm.org/10.1145/2676726.2677009

[56] R. Segala, "Modeling and verification of randomized distributed real-time systems," Massachusetts Institute of Technology, Tech. Rep. MIT/LCS/TR-676, June 1995, ph.D. dissertation.

[57] W. Khreich, E. Granger, A. Miri, and R. Sabourin, "A survey of techniques for incremental learning of HMM parameters," *Inf. Sci.*, vol. 197, pp. 105–130, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.ins.2012.02.017

[58] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser, "N-variant systems: A secretless framework for security through diversity," in *In Proceedings of the 15th USENIX Security Symposium*, August 2006.

Appendix

A. Matrix Properties After Aggregation

Suppose that function f_n is called within function f_m. The call-transition matrix of f_n is merged into the call-transition matrix of function f_m during aggregation.

For property 1 in Definition 5, consider the first row of function f_m. After one aggregation operation between caller function f_m and callee function f_n, the new sum of the the probabilities of the first row is:

$$\begin{aligned}P^{f_m}_{1st\_row\_after} &= \sum_{k\neq k_{f_n}} P^{t_{f_m}}_{\epsilon k} \\ &+ P^{t_{f_m}}_{\epsilon k_{f_n}} * (\sum_{k\neq k_{\epsilon}'} P^{t_{f_n}}_{\epsilon k}) \\ &+ P^{t_{f_m}}_{\epsilon k_{f_n}} * P^{t_{f_n}}_{\epsilon\epsilon'} * (\sum_k \frac{P^{t_{f_m}}_{j_{f_n}k}}{\sum_l P^{t_{f_m}}_{j_{f_n}l}})\end{aligned} \tag{6}$$

In Equation (6), 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 aggregation. Part 3 includes the added transition probabilities when callee function f_n makes no call.

Equation (6) can be reduced as follows.

$$\begin{aligned}P^{f_m}_{1st\_row\_after} &= \sum_{k\neq k_{f_n}} P^{t_{f_m}}_{\epsilon k} + P^{t_{f_m}}_{\epsilon k_{f_n}} * (\sum_{k\neq k_{\epsilon}'} P^{t_{f_n}}_{\epsilon k}) \\ &+ P^{t_{f_m}}_{\epsilon k_{f_n}} * P^{t_{f_n}}_{\epsilon\epsilon'} * 1 \\ &= \sum_{k\neq k_{f_n}} P^{t_{f_m}}_{\epsilon k} + P^{t_{f_m}}_{\epsilon k_{f_n}} * (\sum_k P^{t_{f_n}}_{\epsilon k}) \\ &= \sum_{k\neq k_{f_n}} P^{t_{f_m}}_{\epsilon k} + P^{t_{f_m}}_{\epsilon k_{f_n}} * 1 \\ &= \sum_k P^{t_{f_m}}_{\epsilon k} = P^{f_m}_{1st\_row\_before} = 1\end{aligned} \tag{7}$$

Thus, property 1 holds for the first row of the aggregated matrix. Similarly, one can show that property 1 holds for the first column.

For property 2 in Definition 5, we consider a call f_x in function f_m, $f_x \neq f_n$. After aggregation, the outgoing probability for f_x is:

$$\begin{aligned}P^{f_x}_{out\_after} &= \sum_{k\neq k_{f_n}} P^{t_{f_m}}_{j_{f_x}k} \\ &+ P^{t_{f_m}}_{j_{f_x}k_{f_n}} * (\sum_{k\neq k_{\epsilon}'} P^{t_{f_n}}_{j_\epsilon k}) \\ &+ P^{t_{f_m}}_{j_{f_x}k_{f_n}} * P^{t_{f_n}}_{j_\epsilon k_{\epsilon}'} * (\sum_k \frac{P^{t_{f_m}}_{j_{f_n}k}}{\sum_l P^{t_{f_m}}_{j_{f_n}l}})\end{aligned} \tag{8}$$

Similarly, in Equation (8), for the right-hand 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 (8) can be reduced as:

$$\begin{aligned}P^{f_x}_{out\_after} &= \sum_{k\neq k_{f_n}} P^{t_{f_m}}_{j_{f_x}k} + P^{t_{f_m}}_{j_{f_x}k_{f_n}} * (\sum_{k\neq k_{\epsilon}'} P^{t_{f_n}}_{j_\epsilon k}) \\ &+ P^{t_{f_m}}_{j_{f_x}k_{f_n}} * P^{t_{f_n}}_{j_\epsilon k_{\epsilon}'} * 1 \\ &= \sum_{k\neq k_{f_n}} P^{t_{f_m}}_{j_{f_x}k} + P^{t_{f_m}}_{j_{f_x}k_{f_n}} * (\sum_k P^{t_{f_n}}_{j_\epsilon k}) \\ &= \sum_{k\neq k_{f_n}} P^{t_{f_m}}_{j_{f_x}k} + P^{t_{f_m}}_{j_{f_x}k_{f_n}} * 1 \\ &= \sum_k P^{t_{f_m}}_{j_{f_x}k} = P^{f_x}_{out\_before}\end{aligned} \tag{9}$$

Similarly, for incoming probabilities we have $P^{f_x}_{in\_after} = P^{f_x}_{in\_before}$. Thus, property 2 holds for aggregated matrix.