A Sharper Sense of Self: Probabilistic Reasoning of Program Behaviors for Anomaly Detection with Context Sensitivity

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

***Abstract*—Program anomaly detection models legitimate behaviors of complex software and detects deviations during execution. Behavior deviations may be caused by malicious exploits, design flaws, or operational errors. Probabilistic detection computes the likelihood of occurrences of observed call sequences. However, maintaining context sensitivity in detection incurs high modeling complexity and runtime overhead. We present a new anomaly-based detection technique that is both probabilistic and 1-level calling-context sensitive. We describe a matrix representation and clustering-based solution for model reduction, specifically reducing the number of hidden states in a special hidden Markov model whose parameters are initialized with program analysis. Our extensive experimental evaluation confirms the significantly improved detection accuracy and shows that attacker's ability to conduct code-reuse exploits is substantially limited.**

I. Introduction

The increasing stealth in modern exploits demands more accurate and precise program behavior modeling to identify malicious program behaviors. For example, code-reuse attacks use existing libraries and instruction sequences from inside the victim program's memory to conduct malicious activities. This technique allows attackers to bypass stack-based protection and make arbitrary library and system calls.

Attackers use operating-system provided calls to construct exploits. Therefore, modeling the call sequences of a program for anomalous patterns can potentially detect exploits. This approach is referred to as program anomaly detection in the literature [1]. Program anomaly detection typically has two phases of operation, *i)* constructing a model to capture the legitimate behaviors of a program and *ii)* classifying observed behaviors into benign or anomaly categories. Behavior deviations may be caused by malicious exploits, design flaws, or operational errors.

Probabilistic program anomaly detection can compute the likelihood of occurrences of observed call sequences. Several such solutions exist based on hidden Markov model (HMM) (e.g., [2, 3]). The detection provides quantitative measurement for every observed call sequence. Most recently, researchers demonstrated the use of static program analysis in combination with HMM model to boost the detection accuracy [4]. In these HMM models, the hidden states implicitly or explicitly represent the state (or stage) of the execution; the observation symbols correlate to the program events such as system calls made. Hidden Markov models need to be trained with normal program traces. The trained models can then be used to recognize anomalies in new unknown segments.

Existing probabilistic solutions for program anomaly detection are context-insensitive [4]. They do not distinguish the context of an event. Context sensitivity is the ability to recognize different calling context associated with a called function, when monitoring program traces. Context information increases the precision of program behavior modeling.

Various context information can be learned from runtime execution. For example, Sekar *et al.* recorded program counters to distinguish same system calls [5]. They used program counters to represent the state of the execution, while constructing large finite state automata to recognize normal program behaviors. FSA-based program anomaly detection does not support probabilistic reasoning. VtPath model also includes call stack information as context [6].

However, directly applying fine-grained context-sensitive approaches to Markov-based probabilistic models may result in state explosion in the worst case, i.e., the substantial increase of model size to recognize calls with all possible different contexts in a program. Having a large number of states substantially slows down the model convergence and classification, reducing the timeliness of the detection. In addition, as shown by researchers [4], behavioral models that are constructed solely by learning from traces are inadequate, because they may have high false positive rates due to incomplete traces.

We aim to design a program anomaly detection technique that achieves the following goals:

- To probabilistically reason the likelihood of occurrence of call sequences,
- To record and be able to distinguish the calling context of each call,
- To cover both statically feasible and dynamically observed control flows.

Our model detects anomalies by classifying the call sequences during program executions, and the classification is

based on the probabilistic control-flow representation of a program. We propose a new approach for constructing such a behavior model which incorporates both statically and dynamically analyzed information from programs and the execution traces. We also collect and maintain context information for each call observation.

Our analysis tool statically extracts control-flow graph and call graph information from a program, which is transformed into a rigorous context-sensitive and probabilistic representation by our algorithms. With the initialization of the obtained static information, our customized classification model (namely the hidden Markov model) demonstrates much improved model accuracy.

Compared to many existing approaches, our method does not require any binary instrumentation. Our technical contributions are highlighted as follows.

1) We present a program anomaly-based detection technique that enables probabilistic reasoning on the likelihood of occurrences of call sequences. Compared to related solutions, our system supports 1-level calling-context sensitivity. In our work, 1-level calling-context sensitivity refers to the ability to distinguish the names of the immediate caller functions that invoke system or library calls. Distinguishing calling context of a function generates a more expressive behavioral model and improves detection accuracy. We utilize a compact matrix representation for recording and estimating probabilities of context-sensitive call transitions in a program. Values in the call-transition matrix are used to statically enhance the classifier, namely hidden Markov model (HMM), specifically on the initialization of hidden state and probability parameters.
2) We demonstrate the effectiveness of K-means clustering in reducing the size of hidden Markov model and consequently the training time, while maintaining high detection accuracy. This improvement is particularly important for behavioral models with library calls, because of the diverse context associated with library call invocations. The similarity metric used for clustering is computed between two vectors in the call-transition matrix. With hidden states reduced to $\frac{1}{2}$ to $\frac{1}{3}$ of the original numbers through clustering, the convergence of HMM models is shortened substantially. We observe 75% to 89% reduction in the training time.
3) Our extensive experimental evaluation involves a large number of program traces ($>$ 4,000 test cases) and real-world exploits from utility programs and server programs **`proftpd`** and **`nginx`**. Our experimental results show close to three orders of magnitude accuracy improvement for library calls, and 10-time improvement for system calls on average over context-insensitive counterparts. We also show the low number of ROP gadgets in a program under context-sensitive detection, far from being Turing complete [7]. This limited expressiveness increases the difficulty of successful ROP attacks.

We refer to our prototype as **CMarkov**, short for **C**ontext-sensitive **Markov**. Based on the experimental findings, we attribute the improved accuracy of our CMarkov models to the effectiveness of our program-analysis-guided behavior modeling: **i)** an informed set of initial HMM probability values (transition and emission probabilities and probability distribution of hidden states), and a more optimized number of hidden states. **ii)** a stronger enforcement on legitimate system and library calls with context sensitivity in the program behavior model.

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.

II. Technical Challenges and Solution Overview

A. Attack Model

Our approach monitors and enforces a program's behaviors embodied as system and library call sequences. 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, 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.

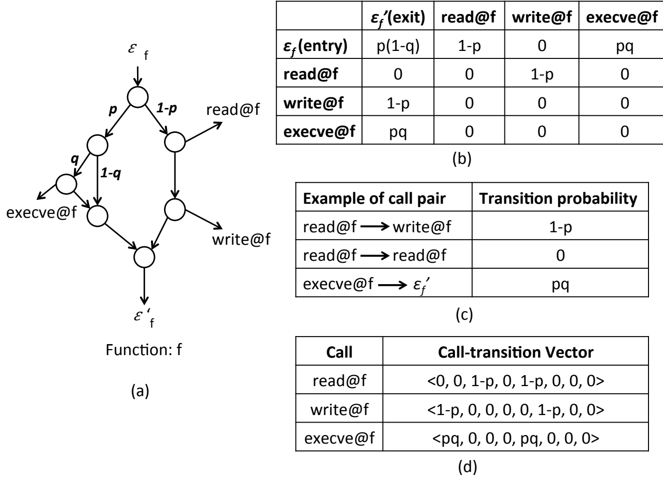
The detection limitation may include advanced mimicry attacks or attack sequences that are extremely short. A hand-crafted mimicry attack was first introduced in [8], where the system calls are made in an order that is compatible with the detection model, but can also perform malicious actions. Although 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.

B. Connecting Markov Model with Control Flow

A hidden Markov model (HMM) probabilistically represents a Markov process consisting of unobserved interconnected hidden states, which emit observable symbols [9] (syscall or libcall in our case).

Definition 1: 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 may be system calls, library calls or user-defined function calls.

The hidden Markov model in our work substantially differs from regular HMM. Regular HMMs arbitrarily choose the hidden states and randomly initialize the probabilities. In contrast,



	ε_f'(exit)	read@f	write@f	execve@f
ε_f(entry)	p(1-q)	1-p	0	pq
read@f	0	0	1-p	0
write@f	1-p	0	0	0
execve@f	pq	0	0	0

(b)

Example of call pair	Transition probability
read@f ⟶ write@f	1-p
read@f ⟶ read@f	0
execve@f ⟶ ε_f'	pq

(c)

Call	Call-transition Vector
read@f	<0, 0, 1-p, 0, 1-p, 0, 0, 0>
write@f	<1-p, 0, 0, 0, 0, 1-p, 0, 0>
execve@f	<pq, 0, 0, 0, pq, 0, 0, 0>

(d)

Fig. 1. Examples of control-flow graph for function $f()$ in (a), call-transition matrix in (b), transition probabilities in (c), and call-transition vectors in (d).

our HMM model is initialized with call transition information extracted from static control-flow analysis. This specialized initialization is achieved with a technique originally proposed in STILO model by Xu *et al.* in [4]. STILO stands for STatically InitiaLized markOv. The STILO approach requires the static control-flow analysis and probability estimation of call transitions (described in CONTEXT IDENTIFICATION and PROBABILITY FORECAST operations in Section III).

The significance of STILO approach is its *one-to-one mapping* between calls made by a program and hidden states in HMM. Its advantage is a more complete coverage of both dynamic and static program behaviors in the classifier. However, their solution is context insensitive.

C. Context Sensitivity

Context-insensitive program anomaly detection models only record names of calls, e.g., **`<call_name>`**. Such 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.

An improvement is to record the caller function of each library or system call as the context environment. An observed call invocation can be represented as **`call_name@caller_function`**. Figure 1 gives such a 1-level calling context-sensitive control-flow graph. We demonstrate the effectiveness and feasibility of such context information in probabilistic program anomaly detection.

In the following example, S_1 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 performs 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 S_2 can be easily identified by a context-sensitive model.

> S_1: normal call sequence:
> ... → read@g → read@f → write@f → execve@g → ...
>
> S_2: abnormal call sequence (code-reuse attack):
> ... → read@g → read@f → write@foo → execve@bar → ...

With calling context, one can distinguish normal call sequence (top) from attack sequence (bottom).

Our solution records 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. Recording context information to other internal functions of a program potentially leads to a more precise representation of the program. However, such a model would incur heavy overhead during both analysis and runtime monitoring. We choose to focus on characterizing library and system calls, not internal functions.

D. Complexity Challenge and State Reduction

Distinguishing calling context substantially increases the size of the behavior model, specifically the number of hidden states in our statically initialized hidden Markov model. To make the models converge in reasonable timeframes (e.g., a day), we perform clustering on the aggregated call-transition matrix (in Def. 5) of a program to identify similar call-transition vectors (i.e., columns or rows in the matrix). The similarity is measured in two aspects: *i)* the sets of incoming and outgoing calls, *ii)* the distribution of probabilities from the incoming and to the outgoing calls. Through clustering, we merge similar call transitions in control-flow graphs and call graphs together, drastically reducing the size of the corresponding hidden Markov model, and thus training time. Details are described in Section III.

III. CLUSTERING-BASED STATE REDUCTION

A. Our Workflow

Major operations in our workflow are described below.

1) CONTEXT IDENTIFICATION: We parse the control-flow graphs of all functions and find the system and library calls. 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 different calls to a function.
2) PROBABILITY FORECAST: Probability information is extracted from control-flow graphs to statically estimate the likelihoods of occurrence for call sequences. We use the control-flow graph of a function to compute a matrix that

represents the likelihoods of occurrence for sequences of calls. We inline these matrices according to the call graph to obtain a (larger) matrix for the entire program.

3) STATE REDUCTION AND INITIALIZATION: The call-transition matrix of the program is used to initialize the parameters of the 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.
State reduction is for models whose numbers of hidden states are large (e.g., > 800). (Without reduction, N is the total number of distinct calls in the program code.) We use K-mean clustering to identify and merge similar calls based on their call-transition vectors (in Def. 6), before they are used to initialize the hidden states in the HMM model. In our prototype, we choose K such that the number of new hidden states is $\frac{1}{3}$ to $\frac{1}{2}$ of the original.
4) TRAIN AND CLASSIFY: We further train the model with normal program traces to adjust the parameters of the HMM classifier, so that it recognizes dynamic program behaviors. For classification, given a segment of program traces (in system call or library call), the model computes the probability of the call segment.

Our model is both flow-sensitive and 1-level calling-context sensitive. The Markov model captures the order of execution of statements in the program. Context information distinguishes system or library calls to the same function name but from different calling contexts (e.g., different call statements).

B. Probability Definitions

We give several types of probabilities 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. They are used to quantify a program's statically inferred control-flow properties, in order to be compatible with parameters of the probabilistic learning model. These definitions follow the STILO model in [4]. We also define the *call-transition matrix of a function* in Definition 5.

Definition 2: 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 its immediate preceding node n_i has just been executed, i.e., $P[n_j|n_i]$.

Definition 3: 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.

Definition 4: The transition probability $P^{t_f}_{ij}$ 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. Each call c in the function is defined and represented as **`call_name@f`** in the context-sensitive model.

Definition 5: Call-transition matrix of a function $f()$ 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 (i, j) stores the likelihood of occurrence for call pair $(c_i \rightarrow c_j)$, i.e., transition probability P^t_{ij}.

Examples of call-transition matrix and transition probability for a context-sensitive CFG are given in Figure 1 (b) and (c), respectively. These defined probabilities are used as follows. Our method first traverses the control-flow graph of each 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, we compute 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 within each function. The transition probabilities are further used to construct a call-transition matrix for each corresponding function.

C. Reduction of Hidden States for Efficiency

A straightforward HMM initialization is to have a one-to-one correlation between system or library calls and hidden states. Because of the many context-sensitive calls, we design a *many-to-one mapping*. That is, a hidden state in HMM may be initialized to represent multiple similar system or library calls. The reduced size of the constructed hidden Markov model helps accelerate the training process. 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.

For the purposes of state reduction through clustering, we define a new *call-transition vector* below. We show examples of call-transition vectors in Figure 1 (d).

Definition 6: For a call-transition matrix of dimension n, a call-transition vector $\vec{x}$ of call c is of size $2n$, consisting of both the transition-from (i.e., column) probabilities of c and the transition-to (i.e., row) probabilities of c.

A straightforward clustering algorithm is to directly compute distances of calls based on their call-transition vectors from Def. 6. The problem of this straightforward clustering algorithm is the overhead of the training process. Training with high-dimension vectors can be expensive, since data points are relatively sparse in the vector space and difficult to model statistically. In our analysis, we use PCA (Principle Component Analysis) to obtain a low-dimension input matrix for clustering. PCA maps data points from a high-dimension space to a low-dimension space, while still preserving distance information in the original data. The input of the PCA is our original matrix with call-transition vectors. The output of the PCA is a reduced-dimension matrix, which is denoted as the post-PCA matrix. For each system or library call c_i, intuitively the *i*-th row of the post-PCA matrix can be viewed as its call transition probabilities in the compact dimension.

We use the K-means clustering algorithm for hidden state reduction. The post-PCA matrix is fed as the input for K-means clustering. We chose the K-means clustering algorithm

Algorithm 1 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 its call-transition vector, which
    is the concatenation of its row of outgoing probabilities and its
    column of incoming probabilities. */
    for (i = 1 → m.size()) do
        vectors[i] = (m[i], m^T[i])
    end for
    /* PCA to reduce dimensions of call-transition vectors*/
    vectors = PCA.transform(vectors)
    /* Cluster vectors. */
    clusters = cluster_vectors(vectors)
    /* Reconstruct clustered call transition matrix*/
    for (j = 1 → 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 its simplicity and efficiency. It is conceivable that advanced clustering algorithms can be used by CMarkov to improve its ability of identifying nodes with similar transitions. The similarity measure used for merging similar call-transition vectors is Euclidean distance computed as $\sqrt{\sum_i^m (x_i - y_i)^2}$ for call-transition vectors $\vec{x}$ and $\vec{y}$ of size m. Intuitively, the similarity of two call vectors is measured by comparing the following: *1)* the sets of incoming and outgoing calls; *2)* the distribution of probabilities from incoming calls and to outgoing calls.

Our initialization of hidden Markov model is based on the clustering results. Specifically, for the similar call-transition vectors appearing in one cluster, we associate the corresponding calls with the same hidden state. Thus, the hidden state has the (averaged) emission probability vector. The transition probability vector associated with that state is also adjusted accordingly. Pseudocode of the state-reduction operation is described in Algorithm 1. This clustering-based transformation is the key to making the context-sensitive probabilistic behavioral modeling become practical.

IV. Probability Forecast with Context-Sensitivity

Context information is maintained and utilized during CFG-based control-flow probability forecast. Key operations in the probability forecast in this context-sensitive model, complexity analyses, and proofs of probability properties extend those of the context-insensitive STILO model.

Computing reachability probability inside each function's control-flow graph is the basis of our probability forecast. To compute reachability probability (Definition 3), our algorithm traverses a CFG and estimates the probability to reach a CFG node from the function entry. We assume that the execution starts from the function entry with the probability of 1.0. The calculation of reachability probabilities starts from the function entry of a CFG, and is performed top down. Formally, for a node n_k in the CFG, the reachability probability P_k^r is computed as in Equation 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 2) for node pair (n_i, n_k).

$$P_k^r = \sum_{\forall\ n_i\ \in\ \texttt{parent set of } n_k} P_i^r * P_{ik}^c \tag{1}$$

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, 11, 12], path frequency [13]. Our prototype uses the uniform distribution. Branch heuristics can be added to further improve our probability-estimation operation [10, 11, 12]. 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.

Computing likelihood of occurrence for call pairs in a function, i.e., transition probability (Definition 4) is as follows. We identify all the nodes $\{L\}$ such that a node $n_l \in L$ satisfies the following three properties. Let n_k be a node in the CFG of $f()$ 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. *iii)* No other nodes on the path between n_k and n_l make any calls. For each node $n_l \in L$, we compute the transition probability $P_{a_k b_l}^{t_f}$ of call pair (c_a, c_b) in $f()$ as Equation (2). With caching, the worst-case complexity of this computation is $O(|E|)$.

$$P_{a_k b_l}^{t_f} = P_k^r * \prod_{i=k}^{l-1} P_{i(i+1)}^c \tag{2}$$

A program may contain multiple functions. Thus, obtaining the call-transition matrix corresponding to the program requires the aggregation of transition probabilities from individual CFG call-transition matrices, which is described next.

Aggregation of call transitions generates a large matrix representation for the entire program. The resulting matrix is compatible with the mathematical representation of a hidden Markov model, and used for initializing the HMM. 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' matrices representation into caller functions', the call graph is needed for the calling relations among functions. We 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 entire program rather than single functions. *ii)* The final transition matrix consists of only system calls or library calls. (Internal function calls are reduced and removed.)

During the aggregation operation, the context information of each call is unchanged. For example, the call **write@f** in the callee function f continued 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 (i.e., 1-level calling-context sensitivity), and this information is maintained throughout the static analysis including the aggregation of call transition matrices. Providing or Maintaining context information at such granularity effectively enforces where a call can actually be made, thus limits the flexibility an attacker may have during exploitation. The worst-case time complexity of this operation is $O(|E|)$.

Our data structure is compact and has low space complexity. The dimension of the aggregated matrix is the number of distinct calls. We do not record the entire call sequences. In addition, all occurrences of the same call pair are added up to one matrix cell. Our design is more efficient than inlining control flow graphs [14].

Summary In CMarkov, the analysis takes control flows that can be statically inferred as inputs, 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 CMarkov HMM model.

V. Experimental Evaluation

Our prototype for the static program analysis is implemented in C/C++ using the `Dyninst` library [15]. 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 [16]) can be used by our implementation in production systems. An acceptable 10% overhead was reported on a hybrid benchmark with realistic workload for auditd [17]. The HMM training and evaluation code is written in Java using the `Jahmm` library [18]. The evaluation or classification of call sequence is relatively efficient. For example, the computation for a 15-call segment takes 0.038 milliseconds (CMarkov 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.

A. 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) [19], as well as an FTP server **proftpd** and an HTTP server **nginx**. These programs average over $52,586$ 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. The coverages of the test cases for programs in SIR are summarized in Table I.

TABLE I
STATISTICS OF PROGRAMS AND TEST CASES USED IN EXPERIMENT. THE COLLECTED TRACES ARE BROKEN INTO 15-GRAMS (SEGMENTS) FOR CLASSIFICATION.

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%

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.

We compare performance of following models:

- *CMarkov:* CMarkov builds its model with statically extracted context-sensitive call transition probabilities, and also goes through dynamic training with context-sensitive call traces.
- *Regular-basic:* This model is the widely accepted HMM-based classification, which is the state-of-the-art probabilistic anomaly detection model (e.g., [2, 3]).
- *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.
- *STILO:* The HMM is initialized with static analysis operations, but without context information, i.e., only call name is recorded for each system or library call.

The accuracy of a regular HMM relies heavily on completeness of training data. Thus, high coverage of test cases in SIR gives the regular HMM a fair chance in the comparison with our 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.

Our experiments answer the following questions.

TABLE II
STATISTICS ABOUT THE CLUSTERING OF CALLS IN THE INITIAL HMM MODELS FOR SELECTED PROGRAMS AND MODELS AND THE ESTIMATED SPEEDUP.

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

1) How much speedup in HMM training is provided by clustering-based state reduction? (In Section V-B)
2) How much improvement in classification accuracy do CMarkov models provide compared to the regular HMM models? (In Section V-C)
3) What are the reasons for CMarkov HMM improvement, and which type of models give more accurate classification, library calls or system calls? (In Section V-C)
4) How does context-sensitive detection limit ROP gadgets? (In Section V-D)
5) Can CMarkov detect real-world attack traces? (In Section V-E)

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 CMarkov 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 (3) and (4), 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.

$$FN = \frac{|\{S_A : P_{S_A} > T\}|}{|\{S_A\}|} \quad (3)$$

$$FP = \frac{|\{S_N : P_{S_N} < T\}|}{|\{S_N\}|} \quad (4)$$

Training and classification are on n-grams of program traces, where n =15 in our experiments (i.e., all segments consist of 15 calls). Researchers found that classification with segments of length 15 produces more precise results than shorter segments [3]. 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 accuracy assessment.

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

B. Clustering for State Reduction

We applied the clustering to programs bash, vim, and proftpd for libcalls. For our evaluation, we choose K as $\frac{1}{2}$ or $\frac{1}{3}$ of the original number of states. Table II shows the reduction of model sizes and estimated training speedup. A substantial 75% to 89% reduction in training time is observed. Despite the reduction in model sizes, our CMarkov models still outperform others as show in the next section.

In another experiment, we trained and tested the libcall HMM for proftpd with unclustered model. The clustered model only needs 10% of the training time, in order to achieve the same false positive rates as its unclustered counterpart.

C. Classification Accuracy

For each program, we compare four different models including CMarkov, STILO, Regular-basic and Regular-context models in their abilities to recognize new segments. New segments include *Normal* segments (through 10-fold cross validation) and *Abnormal-S* segments.

For the Linux utility programs, Figure 2 and Figure 3 give the accuracy comparison among the models. The results show that CMarkov models significantly outperform regular or context-insensitive HMMs in most cases. In addition, CMarkov models work better than STILO models with lower false negative rates.

For the two server programs **proftpd** and **nginx**, we show the library call and system call results in Figure 4 and Figure 5, respectively. Overall, CMarkov models outperform all other regular or context-insensitive models being compared.

Context-sensitive models (including CMarkov and Regular-context) outperform STILO and Regular-basic HMM models by a significant margin, as shown in Figure 4. This phenomenon is partly due to the great diversity of libc calls. Library (libc) calls are usually directly used in user code by

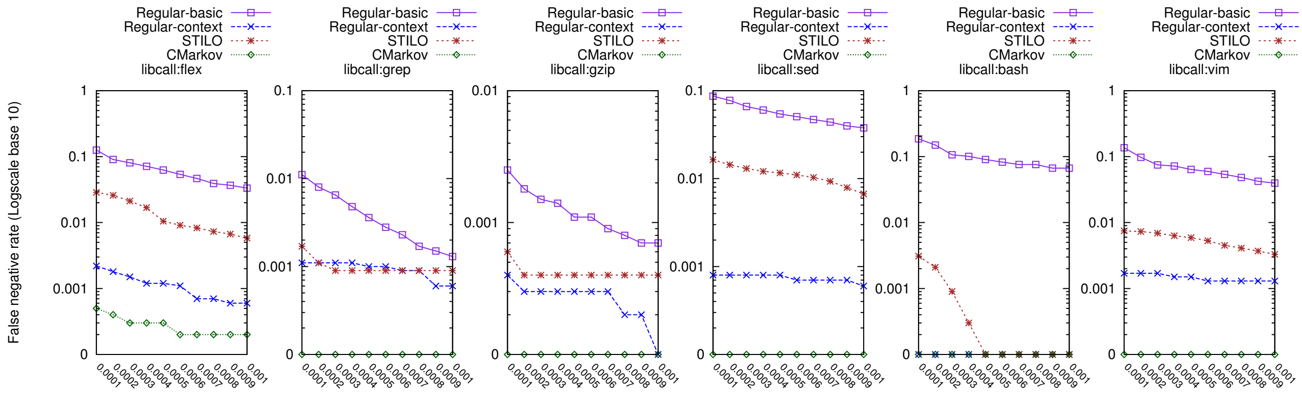


Fig. 2. Comparison of CMarkov, STILO and Regular HMM's false negative rates (in Y-axis, base 10 log scale) for evaluated utility programs on library calls under the same false positive rates (in X-axis). Clustering is applied to the initial models of **proftpd**, **vim** and **bash**.

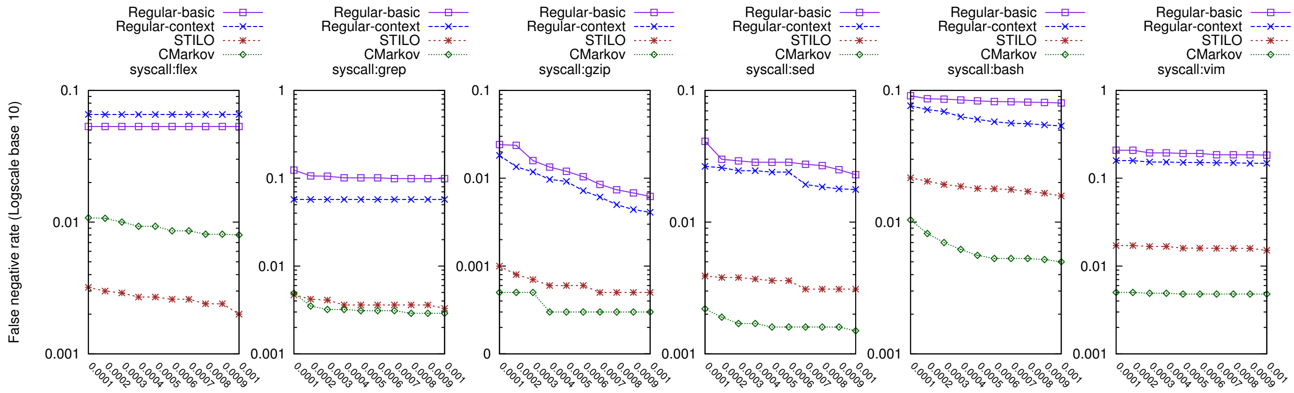


Fig. 3. Comparison of CMarkov, STILO and Regular HMM's false negative rates (in Y-axis, base 10 log scale) for evaluated utility programs on system calls under the same false positive rates (in X-axis).

various internal functions of a program. As a result, the caller context distinguishes library calls within different functions very well. This fact leads to a relatively large number of distinct library calls in our constructed context-sensitive model as well as in the dynamic execution traces.

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 CMarkov and STILO models demonstrate lower false negative rates than the Regular-context and Regular-basic models, as shown in Figure 5. Context-sensitive and context-insensitive models (Regular-basic and Regular-context, STILO and CMarkov) usually have similar numbers of distinct system calls, thus similar numbers of states in the models. As a result their false negative results are very close.

In terms of the *average* detection accuracy on library call traces computed across all evaluated programs, CMarkov gives 452-fold improvement compared to STILO and 31-fold improvement compared to Regular-basic on average. For system call traces, CMarkov has 2-fold improvement compared to STILO on system calls and 10-fold improvement compared

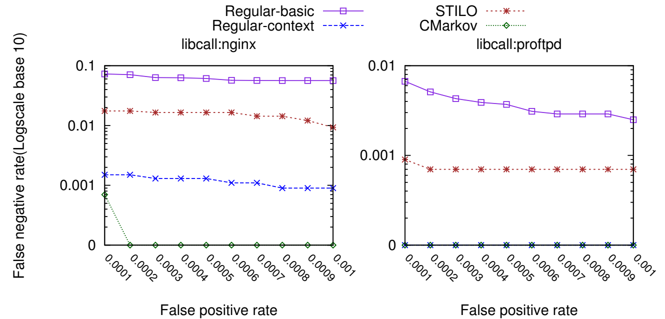


Fig. 4. Comparison of CMarkov, STILO and Regular HMM's false negative rates (in Y-axis, base 10 log scale) for server programs **proftpd** and **nginx** on library calls under the same false positive rates (in X-axis).

to Regular-basic on average. The overall improvement over STILO, especially on libcall traces, confirms the effectiveness of our context-sensitive program anomaly detection.

One of the security advantages of CMarkov is that it imposes strict enforcement on how and where a system call can be made by a program. 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

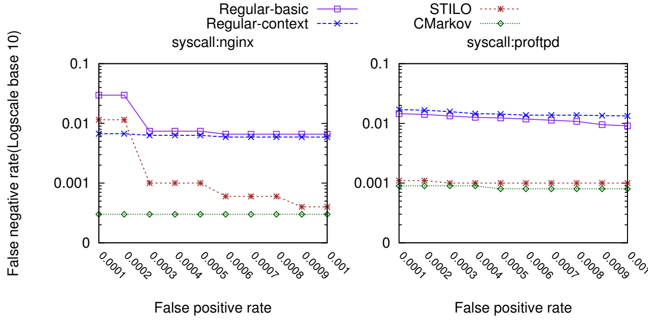


Fig. 5. Comparison of CMarkov, STILO and Regular HMM's false negative rates (in Y-axis, base 10 log scale) for server programs **proftpd** and **nginx** on system calls under the same false positive rates (in X-axis).

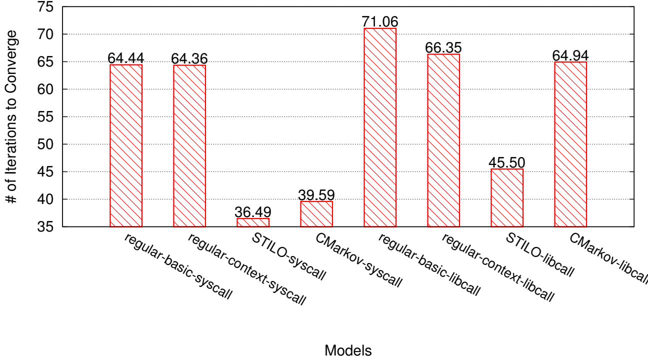


Fig. 6. Average number of training iterations needed for a HMM to converge. Comparisons are among our CMarkov and STILO models and regular HMMs.

within a victim application. Its impact on ROP attack difficulty is discussed in the next section.

For the exceptional case of **`flex`** with system call traces, the CMarkov model shows higher false rates than the STILO model. **`flex`** has the least number of distinct call segments in the termination set (not shown), which indicates less diversity for call sequence patterns in its traces compared to others. Its context-sensitive model may cause possible overfitting.

Convergence Figure 6 gives the average number of iterations for a HMM model to converge. The number is averaged across all eight programs in our experiments. Both CMarkov and STILO HMM models take fewer iterations than the regular HMM models to converge, despite having more states in the models. On average our CMarkov models takes 30% fewer iterations to converge than regular models. CMarkov HMMs take more iterations to converge than STILO HMMs, which is partially due to that the additional context information makes the initial CMarkov HMMs more complex and larger.

D. Limiting ROP Gadgets with Context

We reproduced the following two call segments q_1 and q_2, with the ROP gadgets that exist in the binary of **`gzip`** [2]. Our CMarkov model immediately identifies the incorrect caller context and detects the anomaly. However, the context-insensitive models wrongly classify them as benign (at a false positive rate of 0.0001).

- q_1: [uname, brk, brk, brk, brk, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, read, close, close, unlink, chmod]
- q_2: [brk, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, rt_sigaction, stat, openat, getdents, close, write, read, write, write]

[2] A buffer overflow vulnerability instrumented into **`gzip`** allows us to make further ROP executions.

TABLE III
SEARCH RESULT OF USEFUL [SYSCALL...RET] GADGETS FOR MAKING A SEQUENCE OF SYSTEM CALLS. SEARCH INCLUDES EVALUATED BINARIES AND LIBC.

	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

E. Detection of Real-World Exploits

With the enforcement of caller information, our context-sensitive models successfully detect all the reproduced attacks. For most of the syscall traces, a high percentage of system calls ([30%, 90%]) were found with abnormal caller context information (e.g., either missing or incorrect).

TABLE IV
CMARKOV SUCCESSFULLY DETECTS *Abnormal-A* SEGMENTS FROM REAL-WORLD EXPLOITS.

Vulnerability	Payload
Buffer Overflow (gzip)	*ROP* *ROP_syscall_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*

The attacks evaluated are shown in Table IV and are briefly described next. This experimental setup follows the evaluation in STILO [4]. ROP setup is similar to that in Section V-D. We reproduced a backdoor Trojan (OSVDB-69562) and a buffer overflow (CVE-2010-4221) exploit on a **`proftpd`** server and analyzed the server-side traces. We gave typical attack payloads in the backdoor exploit, which are for establishing various types of communication channels (including telnet, IPv6, TCP, or SSL) between the victim machine and the remote attacker.

F. Summary of Experimental Findings

We summarize our experimental findings below.

TABLE V
ANALYSIS RUNTIME FOR CMARKOV 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.

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

1) **The average classification accuracy of our context-sensitive CMarkov models is orders of magnitudes higher than that of 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. The high classification accuracy in CMarkov model suggests the effectiveness of our static program analysis guided HMM initialization in boosting its performance for anomaly detection.
   Detection with library calls yields 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. 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.
2) **We demonstrate that the available numbers of ROP gadgets in gzip that are compatible with 1-level calling context are low, limiting the success of ROP attacks.** The results under various gadget lengths (2, 6, 10) are shown in Table III.
   CMarkov model detects all the code-injection and subtle code-reuse attacks evaluated. CMarkov also detects carefully prepared ROP-based anomalous system call sequences by identifying their incorrect caller context, whereas the regular HMM model cannot.
3) **Most CMarkov operations can be finished in seconds for the programs evaluated.** The runtime information of CMarkov's analysis operations for library calls and system calls is shown in Table V, including for STATIC CFG CONSTRUCTION, PROBABILITY ESTIMATION, and AGGREGATION OF CALL-TRANSITION MATRIX.
   K-means clustering on library call models reduces the training time by 75% to 89% without compromising detection accuracy.

VI. RELATED WORK

Our discussion is focused on the related control-flow anomaly-detection techniques. We divide them based on the context-sensitive property (i.e., the ability to distinguish calling context at run-time) or the flow-sensitive property (i.e., the ability to analyze the order of statement executions). We refer readers to [20] for a complete and thorough discussion on program anomaly detection literature.

Context-Sensitive Models. The FSA model [5] and VtPath model [6] are both constructed dynamically from program executions. They identify the program counter and return addresses on the stack respectively as the context for each observed system call, which helps improve the precision of their program behavior models. The execution-graph model in [21] 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. However, they share the same issue as other dynamically constructed models where the testing or training data may be incomplete and would thus impair the quality of the learned detection model. Instead of dynamically learning the automaton model from program traces as in [5], one can build a similar flow-sensitive and also context-sensitive automaton by statically analyzing the programs themselves, as shown in the seminal paper by Wagner [22]. The context-sensitive push-down automaton (PDA) (in their *abstract stack* model) in [22], however, may have prohibitive run-time costs.

Giffin *et al.* proposed Dyck model [23] where code is inserted to link the entry and exit of a target function with its call sites for context sensitivity. As a static version of the VtPath, the VPStatic [24] model captures a list of un-returned call site addresses on the stack at the time of each system call. Despite having more accurate program behavior models, the requirement of program instrumentation is still a concern for practical deployment. The IAM (inlined automaton model) in [14] achieves context-sensitivity by inlining every callee function's automata into the caller, which trades more space cost for lower time overhead.

In comparison, CMarkov performs static analysis on the program binaries without any instrumentation. The context information in our model is the caller function of each system or library call, which can be obtained both at static analysis and runtime monitoring. CMarkov has a manageable size controlled at the time of model initialization with statically extracted call transition information. Our model uses the caller function as the context information for each system and libc call, and does not distinguish same calls in a function. The use of program counters as in [5] can further differentiate same system calls made within the same function. Our empirical results show that this fine-grained context does not provide additional detection capability in code reuse attacks.

Consistency in Control Transfer. e-NexSh is a 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 [25]. 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) [26]. 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 [27]. [28] 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 [29]. Total-CFI is a framework for system-wide run-time control-flow integrity enforcement [30] built on a software emulator.

In comparison to CFI techniques, our monitoring system is focused on the call-making portion of the control flow instead of all the execution transfer instructions. We do not require any binary transformation or software emulator. Most CFI systems assume limited dynamic code behaviors [3]; this assumption is not necessary in CMarkov because of our trace-based learning phase. Unlike ours, CFI is not designed to offer any probabilistic behavior analysis.

New attacks against CFI techniques are constantly being reported, e.g., counterfeit object-oriented programming (COOP) has been recently shown to bypass nearly all CFI solutions [31]. Therefore, research efforts on probabilistic program anomaly detection are important, as they can provide complementary security protection to critical systems.

Flow-Sensitive Models. The n-gram models [1, 32, 33] construct a set of all allowable call sequences from the execution traces of a program. It is the simplest flow-sensitive solution. Because the model enumerates all possible call sequences, scalability and efficiency are low.

As a probabilistic learning model, HMM (hidden Markov model) was first presented by Warrender *et al.* [2], and was used to classify program system call sequences for anomaly detection. This is also the model we extensively compare with throughout the paper. By comparing two parallel executions of a same program, Gao *et al.* [34] proposed a HMM-based model that is resilient to their best-estimated mimicry attacks. Different parameters of HMM were systematically studied by the authors in [3] for the impact on model accuracy. These existing HMM-based solutions initialize their models randomly or arbitrarily.

STILO is a HMM model that correlates HMM states with control-flow properties [4]. The model is initialized with the data extracted from static program analysis. STILO is context insensitive. In comparison, CMarkov supports context-sensitive behavioral modeling. Our work addresses the new challenge of state explosion in HMM, specifically, how to support context sensitivity in probabilistic program anomaly detection without incurring heavy computation costs.

[3]E.g., self-modifying code, runtime code generation, and the unanticipated dynamic loading of code [26].

Recently, researchers proposed a machine-learning based detection solution to detect anomalous correlation patterns in execution [35]. Context-insensitive call information is represented in matrices, which are analyzed through clustering and 1-class SVM. *Gu et al.* proposed LEAPS [36] to detect camouflaged attacks with program analysis. This paper is related to our work because it also adopts a preprocess to refine the training data and then deploys statistical learning models to identify benign/malicious call patterns. However, because our goal is to detect code reuse attacks, our approach substantially differs from LEAPS in the following aspects.

1) The preprocess in our approach is used to capture more precise hidden Markov models by statically estimating likelihoods of call sequences. In comparison, the preprocess in LEAPS is used to reduce the noise data and acquire better labeled training dataset.
2) Because of the different detection goals, CMarkov and LEAPS adopt different machine learning techniques. Our approach is based on the hidden Markov model for the purpose of anomaly detection. LEAPS is based on a modified support vector machine for binary classification.

Others. A specialized HMM has been designed for measuring the behavioral distances of two different programs that share similar functionality [34]. This behavior-comparison approach is generally known in the literature as N-variant [37]. How to apply the context-sensitive HMM to N-variant settings is an interesting open problem. Recently, theoretical and abstract anomaly detection frameworks have been proposed to help the security community better define anomalies and understand detection capabilities and limitations. For example, Anceaume *et al.* defined network anomalies with respect to their neighboring environments, and showed that there exist scenarios where isolated and massive anomalies are indistinguishable from a global observer's perspective [38]. Another group of researchers recently proposed a new formal language based framework for comparing the detection capabilities of various anomaly detection techniques [20]. The work also provides abstractions for reasoning the limit of detection accuracy.

Our program anomaly detection solution complements network-centric anomaly detection methods, e.g., LD-Sketch [39], fault-injection based anomaly detection for Service Oriented Architecture (SOA) [40] and trigger-relation based stealthy traffic detection [41]. Jero *et al.* utilizes protocol state machine to detect attacks against transport layer network protocols such as TCP [42]. Buchholz *et al.* showed the use of Markov model for representing and analyzing system availability and dependability [43].

VII. Conclusions and Future Work

In this paper, we presented a HMM-based *probabilistic program anomaly detection* technique that supports 1-level calling context sensitivity. The solution is useful for detecting new and unknown exploits, as well as stealth attacks that alter runtime control flow properties of a program. Our hidden Markov model is specialized with initial probability values extracted through statically analyzing control flows of the

program. We designed and demonstrated a clustering-based method for hidden state reduction. Extensive experimental evaluation with library call and system call sequences of Linux server and utility programs showed 1-3 orders of magnitude improvement over context-insensitive counterparts. Our ongoing work is focused on applying our solutions in order to improve the reliability and dependability of programs on embedded systems in Internet of Things (IoT).

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful comments on the work. This work has been supported by ONR grant N00014-13-1-0016.

References

[1] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *Proc. of S&P*, 1996.

[2] C. Warrender, S. Forrest, and B. A. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *Proc. of S&P*, 1999.

[3] D.-Y. Yeung and Y. Ding, "Host-based intrusion detection using dynamic and static behavioral models," *Pattern Recognition*, 2003.

[4] K. Xu, D. Yao, B. Ryder, and K. Tian, "Probabilistic program modeling for high-precision anomaly classification," in *Proc. of CSF*, 2015.

[5] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proc. of S&P*, 2001.

[6] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proc. of S&P*, 2003.

[7] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. of CCS*, 2007.

[8] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proc. of CCS*, 2002.

[9] L. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, Feb 1989.

[10] T. Ball and J. R. Larus, "Branch prediction for free," in *Proc. of PLDI*, 1993.

[11] 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, 1997.

[12] Y. Wu and J. R. Larus, "Static branch frequency and program profile analysis," in *Proc. of MICRO*, 1994.

[13] R. P. L. Buse and W. Weimer, "The road not taken: Estimating path execution frequency statically," in *Proc. of ICSE*, 2009.

[14] R. Gopalakrishna, E. H. Spafford, and J. Vitek, "Efficient intrusion detection using automaton inlining," in *Proc. of S&P*, 2005.

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

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

[17] Cost of Security. http://institute.lanl.gov/isti/summer-school/cluster_network/projects-2011/2011%20Yellow%20Team_Lopez%20Mortensen%20Chambers.pdf.

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

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

[20] X. Shu, D. Yao, and B. Ryder, "A formal framework for program anomaly detection," in *Proc. of RAID*, 2015.

[21] D. Gao, M. K. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection," in *Proc. of CCS*, 2004.

[22] D. Wagner and D. Dean, "Intrusion detection via static analysis," in *Proc. of S&P*, 2001.

[23] J. T. Giffin, S. Jha, and B. P. Miller, "Efficient context-sensitive intrusion detection," in *Proc. of NDSS*, 2004.

[24] 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 *Proc. of S&P*, 2004.

[25] G. S. Kc and A. D. Keromytis, "e-NeXSh: Achieving an effectively non-executable stack and heap via system-call policing," in *Proc. of ACSAC*, 2005.

[26] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity: Principles, implementations, and applications," in *Proc. of CCS*, 2005.

[27] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *Proc. of CCS*, 2011.

[28] 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 *Proc. of S&P*, 2013.

[29] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proc. of USENIX Security*, 2013.

[30] A. Prakash, H. Yin, and Z. Liang, "Enforcing system-wide control flow integrity for exploit detection and diagnosis," in *Proc. of AsiaCCS*, 2013.

[31] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proc. of S&P*, 2015.

[32] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, 1998.

[33] C. Wressnegger, G. Schwenk, D. Arp, and K. Rieck, "A close look on N-grams in intrusion detection: Anomaly detection vs. classification," in *Proc. of AISec*, 2013.

[34] D. Gao, M. K. Reiter, and D. X. Song, "Beyond output voting: Detecting compromised replicas using HMM-based behavioral distance," *IEEE TDSC*, 2009.

[35] X. Shu, D. Yao, and N. Ramakrishnan, "Unearthing stealthy program attacks buried in extremely long execution paths," in *Proc. of CCS*, 2015.

[36] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang, and D. Xu, "LEAPS: Detecting camouflaged attacks with statistical learning guided by program analysis," in *Proc. of DSN*, 2015.

[37] 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 *Proc. of USENIX Security*, 2006.

[38] E. Anceaume, Y. Busnel, E. L. Merrer, R. Ludinard, J. L. Marchand, and B. Sericola, "Anomaly characterization in large scale networks," in *Proc. of DSN*, 2014.

[39] Q. Huang and P. P. C. Lee, "Ld-sketch: A distributed sketching design for accurate and scalable anomaly detection in network data streams," in *Proc. of INFOCOM*, 2014.

[40] A. Ceccarelli, T. Zoppi, A. Bondavalli, F. Duchi, and G. Vella, "A testbed for evaluating anomaly detection monitors through fault injection," in *Proc. of ISORC*, 2014.

[41] H. Zhang, D. D. Yao, and N. Ramakrishnan, "Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery," in *Proc. of AsiaCCS*, 2014.

[42] S. Jero, H. Lee, and C. Nita-Rotaru, "Leveraging state information for automated attack discovery in transport protocol implementations," in *Proc. of DSN*, 2015.

[43] P. Buchholz and J. Kriege, "Markov modeling of availability and unavailability data," in *Proc. of EDCC*, 2014.