Long-Span Program Behavior Modeling and Attack Detection

XIAOKUI SHU, IBM Research
DANFENG (DAPHNE) YAO, Virginia Tech
NAREN RAMAKRISHNAN, Virginia Tech
TRENT JAEGER, Pennsylvania State University

Intertwined developments between program attacks and defenses witness the evolution of program anomaly detection methods. Emerging categories of program attacks, e.g., non-control data attacks and data-oriented programming, are able to comply with normal trace patterns at local views. This paper points out the deficiency of existing program anomaly detection models against new attacks and presents long-span behavior anomaly detection (LAD), a model based on mildly context-sensitive grammar verification. The key feature of LAD is its reasoning of correlations among arbitrary events occurred in long program traces. It extends existing correlation analysis between events at a stack snapshot, e.g., paired `call` and `ret`, to correlation analysis among events historically occurred during the execution. The proposed method leverages specialized machine learning techniques to probe normal program behavior boundaries in vast high-dimensional detection space. Its two-stage modeling/detection design analyzes event correlation at both binary and quantitative levels. Our prototype successfully detects all reproduced real-world attacks against `sshd`, `libpcre`, and `sendmail`. The detection procedure incurs 0.1~1.3 ms overhead to profile and analyze a single behavior instance that consists of tens of thousands of function call or system call events.

CCS Concepts: • **Security and privacy → Intrusion detection systems**; *Formal methods and theory of security*; *Systems security*;

Additional Key Words and Phrases: Intrusion detection, program analysis, context-sensitive grammar, co-occurrence analysis, event frequency correlation, machine learning

ACM Reference format:
Xiaokui Shu, Danfeng (Daphne) Yao, Naren Ramakrishnan, and Trent Jaeger. 2017. Long-Span Program Behavior Modeling and Attack Detection. *ACM Trans. Priv. Sec.* 1, 1, Article 1 (August 2017), 28 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

Attacks to programs are one of the oldest and fundamental threats to computing systems, which evolve and constitute latest attack vectors and advanced persistent threats. Anomaly-based intrusion detection discovers aberrant executions caused by attacks, misconfigurations, program bugs, and unusual usage patterns. The approach models normal program behaviors instead of the threats. It does not bear time lags between emerging attacks and deployed countermeasures as standard

This work has been supported by ONR grant N00014-13-1-0016, AFRL and DARPA grant FA8650-15-C-7561, and ARO YIP W911NF-14-1-0535. A preliminary version of the work appeared in the Proceedings of the 2015 ACM Conference on Computer and Communications Security (CCS) [53].

2471-2566/2017/8-ART1 $15.00
https://doi.org/0000001.0000001

defenses do, which are built upon retrospects of inspected attacks. The merit of program anomaly detection is its independence from attack signatures. This property potentially enables proactive defenses against new and unknown threats.

The detection accuracy of program anomaly detection methods relies on the precision of normal program behavior description and the completeness of the training [54]. Primitive program attacks, e.g., return addresses manipulation and library/system call injection, result in great variation from normal behaviors. Thus, they can be distinguished from relatively imprecise descriptions of normal program behaviors, e.g., *n*-gram system call anomaly detection [12].

Program anomaly detection models evolve as attacks share more and more similarities with normal behaviors. Attacks using explicit control flow manipulation, e.g., tampering with return addresses on the stack, are no longer effective due to the deployment of standard defense mechanisms such as non-executable stack, address space layout randomization, and control-flow integrity. New program attacks utilize indirect means of control flow manipulation, e.g., data-oriented programming [25], or abuse programs within legal control flows, e.g., denial of service attacks. The emerging stealthy attacks diminish the effectiveness of existing anomaly-based intrusion detection models to distinguish them from normal program executions.

Take the remote authentication subroutine in `sshd` as an example, an attacker can launch a stealthy attack by overwriting a flag variable recording authentication results in the authentication subroutine. The overwriting prior to specific authentication operations enables the attacker to bypass critical security control and log in after a failed authentication. There is no illegal control flow or short abnormal system call sequences in this attack. Therefore, it is difficult for existing models to detect. The `sshd` attack involves breaches of data integrity for altering the execution path. However, this is not necessary. Denial of service attacks and some data retrieval attacks leverage legal-but-unusual execution patterns to fulfill effective attacks. Such attacks can only be detected by existing anomaly detection methods to some extent.

Program anomaly detection utilizes execution traces to describe program behaviors and search for anomalous behaviors [52]. They can be abstracted as former grammar parsers to validate program traces, and existing models are restricted to regular grammar parsers [11, 12, 28] and context-free grammar parsers [10, 19, 30, 49]. Some comprehensive approaches extend deterministic language parsers to probabilistic ones [15, 22, 36, 63], verify some additional data-flow information [3, 18, 19, 38] or perform simple frequency analysis [14, 15, 61]. However, none of them is designed to represent a context-sensitive grammar parser, which is able to better recognize normal program behaviors and detect aforementioned attacks.

The key difference between a context-sensitive grammar model and existing (regular or context-free grammar) models is *the ability to correlate arbitrary events in a long program trace*. Two segments of code executed during a run may correlate by specific control or data links. Context-sensitive grammar parsing enables the mining of event correlations in long program traces. The correlations reveal specific behavior properties of normal executions, which could distinguish them from anomalous runs and attacks. For instance, the aforementioned `sshd` attack breaks the correlation between the authentication result and the login operation.

The complete context-sensitive analysis is impractical to reach by a real-world detection tool due to non-polynomial training complexity [54]. However, it is possible to construct parsers that recognize context-sensitive properties in the traces yet avoid the impractical complexity of complete context-sensitive grammars.

We present *long-span behavior anomaly detection* (LAD) – a mildly context-sensitive grammar program anomaly detection model to characterize program behaviors and detect stealthy attacks. We approach the anomaly detection problem from the machine learning perspective and blueprint a two-stage data mining solution for event correlation analysis in long program traces. LAD can be abstracted as a stochastic mildly context-sensitive language parser. It consists of a training phase (the stochastic language construction procedure) and a detection phase (new behavior/trace testing). The context-sensitive property enables the model to learn correlations between events that are millions of events away in the trace. The stochastic language estimates proper boundaries of normal behaviors in a high-dimensional detection space and mitigates potential oversensitive detection due to insufficient training samples.

Our approach overcomes two major challenges in constructing a context-sensitive grammar model for program anomaly detection.

Training scalability challenge: the complete event reasoning in extremely long trace segments forms a vast detection space, the size of which is exponential to the length of the training trace segments. It requires exponential training time to converge. Insufficient training results in high false positive rates as tested in many n-gram methods with $n > 40$ for long segment analysis. Our approach reduces the potential exponential-size space to a constant-size high-dimensional detection space by profiling program behavior instances into fixed size matrices. The matrix profiles are initialized by static program analysis and filled by program behavior instances (dynamic traces).

Behavior diversity challenge: various functionalities in real-world programs lead to diverse program behaviors. Diverse normal data points in the high-dimensional detection space make it difficult to seek a hyperplane in traditional classifiers, e.g., one-class SVM, for precisely distinguishing anomalous behaviors from normal ones. Stealthy attacks can often exploit the imprecision of normal boundaries generated by traditional classifiers and subvert the detection. Our two-stage design recognizes diverse normal behaviors in clusters and performs precise characterizations of normal behaviors inside each cluster.

The contributions of our work are summarized as follows.

- We identify the need to correlate events in long program traces for the detection of stealthy program attacks that alter execution paths instead of control flows. We formalize a stochastic mildly context-sensitive language model for mining arbitrary event correlations in long program traces and estimating normal boundaries in a high-dimensional detection space.
- We embody the proposed mildly context-sensitive language model with a two-stage data mining approach, which consists of a *constrained* agglomerative clustering algorithm for addressing the behavior diversity challenge and a combination of probabilistic and deterministic models for precise intra-cluster behavior modeling. The two-stage detection approach mines event correlations from fixed-size matrix profiles of program behaviors, which addresses the training scalability challenge.
- We prototype our program anomaly detection approach on Linux and evaluate its detection capability, accuracy, and performance with `sshd`, `libpcre`, and `sendmail`. Our prototype is trained with over 22,000 normal profiles and detect over 800 reproduced real-world attacks. High detection accuracy is demonstrated against four categories of synthetic anomalies, and the testing of a single program behavior profile with 1k to 50k function/system call events incurs only 0.1~1.3 ms.

2 SECURITY MODEL

We discuss stealthy attacks that do not directly alter control flows, point out the need of event correlation analysis in long program traces, and present a mildly context-sensitive grammar parser for the detection of such attacks.

```
 1: void do_authentication(char *user, . . . ) {
 2:   int authenticated = 0;
      . . .
 3:   while (!authenticated) {
 4:     type = packet_read();
 5:     switch (type) {
          . . .
 6:       case SSH_CMSG_AUTH_PASSWORD:
            . . .
 7:         if (auth_password(user, password)) {
 8:           memset(password, 0, strlen(password));
 9:           xfree(password);
10:           log_msg(". . . ", user);
11:           authenticated = 1;
12:           break;
            }
13:         memset(password, 0, strlen(password));
14:         debug(". . . ", user);
15:         xfree(password);
16:         break;
          . . .
        }
17:     if (authenticated) break;
        . . .
```

Fig. 1. sshd flag variable overwritten attack.

2.1 Aberrant Path Attack

We study a category of stealthy attacks – *aberrant path attacks*, which contain infeasible, inconsistent, or aberrant execution paths, but they obey legitimate control-flow graphs. Aberrant path attacks can evade existing detection mechanisms because of the following properties of the attacks:

- not conflicting with control-flow graphs
- not incurring anomalous call arguments
- not introducing unknown short call sequences

Aberrant path attacks are realistic threats and gain in popularity since primitive attacks are less effective due to memory protection, e.g., address space layout randomization (ASLR) [50]. A popular example is the `sshd` flag variable overwritten attack first described by Chen et al. [6]. The attack takes advantage of an integer overflow vulnerability found in several implementations of the SSH1 protocol [35]. Illustrated in Figure 1, an attacker can overwrite the flag integer `authenticated` when the vulnerable procedure `packet_read()` is called. If `authenticated` is overwritten to a nonzero value, line 17 is always `True` and `auth_password()` on line 7 is no longer effective.

Aberrant path attacks do not only allow the attacker to bypass authentication procedures, but can also be leveraged to cover Turing-complete operations as control flow manipulation attacks. We list four common types of aberrant path attacks as follows.

(1) *Non-control data attacks* that reorganize existing control flows to create new execution paths fall into the categories of aberrant path attacks[1]. The attacks hijack programs without manipulating their control data (data loaded into program counter, e.g., return addresses). It is named by Chen et al. [6] and generalized by Hu as *data-oriented programming* [24, 25]. Hu utilizes compromised variables to construct loop trampolines and achieve Turing-complete functionalities without directly tampering control data.
(2) *Workflow violation attacks* exploit weak workflow enforcements of a system. The attacker usually executes part of the program to bypass access control [7], leak critical information, or disable a service (e.g., trigger a deadlock). One example is *presentation layer access control bypass* in web applications. If the authentication is only enforced at the presentation layer, which is not securely coupled to its underlying business logic layer, an attacker can directly access the business logic layer and read/write data.
(3) *Exploit preparation* is a common step preceding the launch of an exploit payload. It usually utilizes *legal control flows* to load essential libraries, arranges memory space (e.g., heap feng shui [55]), seeks addresses of useful code and data fragments (e.g., ASLR probing [50]), and/or triggers particular race conditions.
(4) *Service abuse attacks* do not take control of a program. Instead, the attacks utilize *legal control flows* to compromise the availability (e.g., Denial of Service attack), confidentiality (e.g., Heartbleed data leak [23]), and financial interest (e.g., click fraud) of target services.

2.2 Security Goal: Event Correlation Mining in Long Program Traces

The key to the detection of aberrant path attacks is the knowledge of instruction correlations in a long program trace. Even legal control flows are preserved, the execution paths of the program under attacks are different than normal. Therefore, the attacker can achieve their attack goals.

Taking the `sshd` flag variable overwritten attack (Figure 1) as an example, normal execution paths should contain line 8-12 if line 17 is `True` (a shell is spawned afterward). The attack trace, on the contrary, contains the shell spawn procedure (line 18-) without the the execution of line 8-12. Co-occurrence between instructions in an execution can be analyzed to detect such an attack.

Another example is the loop trampoline proposed by Hu to construct a Turing-complete non-control data attack [25]. In this attack, a vulnerable variable in a loop branch is exploited to prepare a gadget – a fraction of the malicious payload. The attacker executes the loop trampoline at an unusual pattern to chain a string of gadgets together and achieve Turing-complete operations. The attack reflects an unusual pattern of the loop usage because of the gadget construction. It could also result in broken co-occurrence among instructions in normal runs.

The goal of the detection is to *learn event correlations in long program traces*. Events refer to function calls, jumps, or even generic instructions in a program trace. The correlations append additional information to control flows, making it possible to enforce co-occurrences of branches as well as different portions of a process.

We define event correlation in two levels.

(1) *Event co-occurrence* denotes the binary relation between co-occurred events in a long program trace. Repeating events should be deduplicated, and event appearances are studied in binary forms.
(2) *Event occurrence frequency relation* denotes quantitative relation among event occurrence frequencies in a long program trace. It provides more fine-grained knowledge about the execution path than event co-occurrences.

[1]Simple non-control data attacks that only exploit arguments of legitimate `exec` system calls may not result in explicit aberrant path attacks.

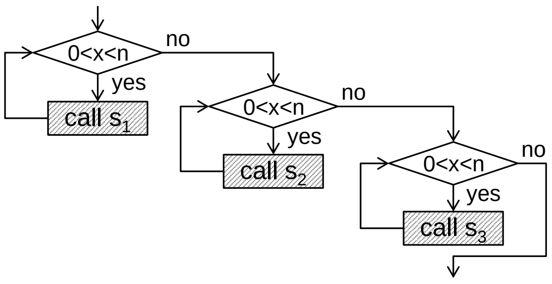


Fig. 2. An example of event relation restricted by induction variables.

2.3 Security Model: A Stochastic Mildly Context-Sensitive Language

We first define a program trace as the building block to formalize program anomaly detection models. Then we present a mildly context-sensitive language model for characterizing event co-occurrences and event frequency relations in a long program trace.

Definition 2.1. A program trace T is the sequence of all instructions executed in an autonomous portion of a program execution.

T records the sequence of all executed instructions (addresses and arguments). In real-world detection systems, select instructions are used to represent all instructions in the construction of T. The simplification sacrifices precision for practicality, e.g., smaller tracing and modeling overhead. For example, T only contains system call instructions, e.g., SYSENTER, in system call anomaly detection systems.

To build practical detection systems, T can be a long portion of a finite or infinite[2] program trace. In practice, T is partitioned from an entire trace based on boundaries of program functionalities to represent program behaviors with semantic meanings. Examples of partitioning for security analysis include *i)* critical subroutine call/return, *ii)* threads creation/termination, or *iii)* the entire execution of a small program.

Given a set of normal program traces (generated from dynamic or static/synthetic program analysis), a program anomaly detection system forms a formal grammar to accept all normal traces and reject others. The detection system can be written as the set of all normal program traces, i.e., a formal language: $L = \{T \mid T \text{ is normal}\}$.

To fulfill the security goal of describing arbitrary event correlations in long program traces, one needs a context-sensitive language model to parse program traces.

THEOREM 2.2. *A program anomaly detection model describing arbitrary event correlations in a long program trace is a context-sensitive language model.*

PROOF. We prove the contrapositive of Theorem 2.2: a context-free language cannot model arbitrary event correlation in a long program trace.

Figure 2 illustrates a simple block of code that results in a strong relation of the occurrence frequencies of system calls s_1, s_2 and s_3 in an execution trace. The traces of the program can be described as a language $\dot{L} = \{\cdots s_1^n s_2^n s_3^n \cdots\} = \{\cdots s_1 s_2 s_3 \cdots, \cdots s_1 s_1 s_2 s_2 s_3 s_3 \cdots, \ldots\}$. It is proved that the formal language $\ddot{L} = \{s_1^n s_2^n s_3^n\}$ is not a context-free language – $\ddot{L}$ does not satisfy the *pumping lemma* property shared among all context-free languages [2]. Therefore, $\dot{L}$ (a superset of $\ddot{L}$) is not a context-free language. □

[2]A program could run continuously and yield an infinite trace.

Table 1. An Example of $\tilde{L}$

$\Sigma = \{s_1, s_2, s_3\}$	program behavior modeled by three system calls
$T \in \Sigma^*$	a trace is an arbitrary combination of items in Σ
$f_i = T.count(s_i) \mid 1 \le i \le 3$	f_i represents the frequency of s_i in T
$P(T) = \begin{cases} 0 & f_1 \neq f_2 \\ min(f_3/f_1, 1) & f_1 = f_2 \end{cases}$	probability generation
$\eta = 0.5$	string probability threshold

The detection system in this example accepts program traces with equal numbers of s_1 and s_2 and some s_3 that occur more than half times of s_1.

Long program traces modeled as strings establish a high-dimensional space for modeling, which makes the training of a generic context-sensitive language model impractical. When processed as strings, each position in the traces, e.g., the i^{th} item, establishes one dimension in the detection space. The dimensionality of the space equals to the length of the longest trace.

To build a practical detection mechanism, we modify the generic context-sensitive language to avoid *exponential training convergence time* and *oversensitive detection caused by sampled normal behaviors in training*. The former refers to the exponential number of traces/objects to model regarding the length of the traces. The latter refers to false positives caused by unsampled normal behaviors/traces that are recognized incorrectly by the detection system.

- The key to our non-exponential time solution is to focus on the event relational information other than the order information among events in traces. We build a restricted context-sensitive language that only characterizes the relation quantitatively among events, but not the order of events in long traces. Our language is a generalization of the context-sensitive language *Bach* [47]. The latter only characterizes strings with equally occurred symbols.
- Our solution to the oversensitive detection issue is a stochastic language $\tilde{L} = \{T \mid P(T) > \eta\}$, which estimates proper boundaries of normal behaviors from training samples. In our stochastic language, each tested trace T yields a probability showing how likely T can be accepted by $\tilde{L}$. A threshold $\eta \in [0, 1]$ is used to decide the acceptance of the trace by $\tilde{L}$.

Our program anomaly detection model is defined as a stochastic mildly context-sensitive language $\tilde{L}$ in Definition 2.3.

Definition 2.3. $\tilde{L} = \{T|(T \in \Sigma^*) \wedge (P(T) > \eta) \wedge (S \Rightarrow^+ T)\}$ where $\Sigma = \{e_0, e_1, e_2, \ldots\}$ is the set of all events monitored; $S \Rightarrow^+ T$ is the string generation operation that appends a symbol to the left or right of an existing string/trace; all strings start from an empty string $S = \{\epsilon\}$; the acceptance of a string/trace is checked by $P(T) > \eta$, which verifies event frequency relation in a string.

We give an example of $\tilde{L}$ in Table 1. The detection system accepts program traces as normal when *i)* the trace contains same numbers of s_1 and s_2, and *ii)* the number of s_3 in the trace is higher than half of the number of s_1. The acceptance rule of a specific language is customized by the $P(T)$ function, which can characterize both the event co-occurrence relation and the event occurrence frequency relation.

3 LAD SYSTEM OVERVIEW

We present long-span behavior anomaly detection (LAD), a detection system to embody the stochastic mildly context-sensitive language $\tilde{L}$ discussed in Section 2.3. The goal of the detection system is to fulfill both the (binary) event co-occurrence analysis and the (quantitative) event occurrence frequency analysis supported by $\tilde{L}$.

We present an overview of LAD for analyzing long program traces in this section. We develop a constrained agglomerative clustering algorithm to overcome the behavior diversity challenge. We develop a compact and fixed-length matrix representation to profile long program traces. The matrix representation only records relational information among events. It is an implementation to avoid *exponential training convergence time* as discussed in $\tilde{L}$.

3.1 Profiling Program Behaviors

We describe the profiling procedure in LAD using the example event set: {`call`, `ret`[3]}[4]. `call` and `ret` expose user-space program activities and provide a more detailed understanding of the program execution than system calls. Function calls are exposed in some automata-based methods, e.g., Dyck model [19], but for a different purpose, i.e., eliminating non-deterministic paths.

We denote the overall activity of a program within a long trace segment T[5] as a behavior instance b. Instance b recorded in T is profiled in two matrices:

Definition 3.1. An event co-occurrence matrix O is an $m \times n$ Boolean matrix recording co-occurred call events in a behavior instance b. $o_{i,j} =$ `True` indicates the occurrence of the call from the i-th row symbol (a routine) to the j-th column symbol (a routine). Otherwise, $o_{i,j} =$ `False`.

Definition 3.2. A transition frequency matrix F is an $m \times n$ nonnegative matrix containing occurrence frequencies of all calls in a behavior instance b. $f_{i,j}$ records the occurrence frequency of the call from the i-th row symbol (a routine) to the j-th column symbol (a routine). $f_{i,j} = 0$ if the corresponding call does not occur in O.

$$O_{m,n} = \begin{bmatrix} o_{1,1} & o_{1,2} & \cdots & o_{1,n} \\ o_{2,1} & o_{2,2} & \cdots & o_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ o_{m,1} & o_{m,2} & \cdots & o_{m,n} \end{bmatrix} \qquad F_{m,n} = \begin{bmatrix} f_{1,1} & f_{1,2} & \cdots & f_{1,n} \\ f_{2,1} & f_{2,2} & \cdots & f_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ f_{m,1} & f_{m,2} & \cdots & f_{m,n} \end{bmatrix}$$

For one specific b, O is a Boolean interpretation of F that

$$o_{i,j} = \begin{cases} \texttt{True} & \text{if } f_{i,j} > 0 \\ \texttt{False} & \text{if } f_{i,j} = 0 \end{cases} \tag{1}$$

O and F are succinct representations of the *dynamic call graph* of a running program. m and n are total numbers of possible callers and callees in the program, respectively. Row/column symbols in O and F are determined through static analysis. m may not be equal to n, in particular when calls inside libraries are not counted.

Bitwise operations, such as `AND`, `OR`, and `XOR` apply to co-occurrence matrices. For example, O' `AND` O'' computes a new O that $o_{i,j} = o'_{i,j}$ `AND` $o''_{i,j}$.

Profiles at different granularities Although designed to be capable of modeling user-space program activities via function calls, LAD can also digest coarse level program traces for learning program behaviors. For example, system calls can be traced and profiled into O and F to avoid

[3]`ret` is paired with `call`, which can be verified via existing CFI technique. We do not involve the duplicated correlation analysis of `ret` in our model, but we trace `ret` to mark function boundaries for long trace partitioning (discussed in Section 5 and Section 6.1).

[4]Σ is larger than the set of two because `call`/`ret` with different addresses are different events/symbols in Σ.

[5]The definition of T should be specified by security analysts or software developers. Automatic definition construction could be developed to accelerate the deployment of the system.

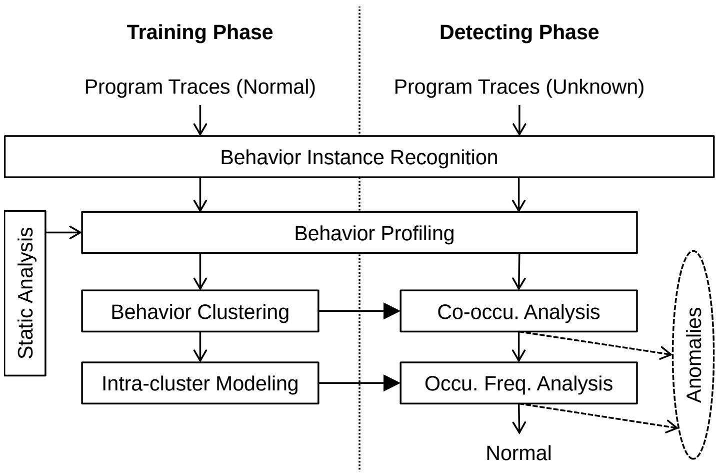


Fig. 3. Information flows among operations in two stages and two phases of LAD.

excessive tracing overheads in performance-sensitive deployments. The semantics of the matrices changes in this case; each cell in O and F represents a statistical relation between two system calls. The detection is not as accurate as our standard design because system calls are coarse descriptions of program executions.

3.2 Architecture

LAD consists of two complementary stages of modeling and detection for event co-occurrence analysis and event occurrence frequency analysis, respectively.
The first stage models the binary representation of event co-occurrences in a long program trace via event co-occurrence matrix O for *event co-occurrence analysis*. It consists of a training operation Behavior Clustering and a detection operation Co-occurrence Analysis.
The second stage models the quantitative frequency relation among events in a long trace via transition frequency matrix F for *event occurrence frequency analysis*. It consists of a training operation Intra-cluster Modeling and a detection operation Occurrence Frequency Analysis.

We illustrate the architecture of LAD in Figure 3 and brief each operation below.

1. Behavior Profiling recognizes target long trace segments $\{T_1, T_2, \ldots\}$ in raw traces and profiles b from each T into O and F. Symbols in F and O are retrieved via static program analysis or system call table lookup.
2. Behavior Clustering is a training operation. It takes in all normal behavior instances $\{b_1, b_2, \ldots\}$ and outputs a set of behavior clusters $\mathbb{C} = \{C_1, C_2, \ldots\}$ where $C_i = \{b_{i_1}, b_{i_2}, \ldots\}$ and $b_{i\_} \in \{b_1, b_2, \ldots\}$.
3. Intra-cluster Modeling is a training operation. It is performed in each cluster. It takes in all normal behavior instances $\{b_{i_1}, b_{i_2}, \ldots\}$ for C_i and constructs one deterministic model and one probabilistic model for computing the refined normal boundary in C_i.
4. Co-occurrence Analysis is an inter-cluster detection operation that analyzes O (of b) against clusters in $\mathbb{C}$. If behavior instance b is normal, it reduces the detection problem to subproblems within a set of behavior clusters $\mathbb{C}_b = \{C_{b_1}, C_{b_2}, \ldots\}$, in which b closely fits.

5. Occurrence Frequency Analysis is an intra-cluster detection operation that analyzes F (of b) in each C_b. Behavior instance b is normal if F abides by the rules extracted from C_b and F is within the normal boundary established in C_b.

4 INTER-/INTRA-CLUSTER DETECTION

We detail the two stages of the training and modeling procedures in LAD. The first stage contains a constrained clustering algorithm, which differentiates diverse program behaviors, divides the detection problem into subproblems, and performs the first round dimensionality reduction. The clustering enables inter-/intra-cluster detection in the first/second stage, respectively.

4.1 Behavior Clustering (Training)

We develop a constrained agglomerative clustering algorithm that addresses two special needs to handle program behavior instances for anomaly detection: *i)* long tail elimination, and *ii)* borderline behavior treatment. Agglomerative clustering is a bottom-up hierarchical clustering strategy that merges nearest clusters till stopping criteria, e.g., distance of clusters, number of clusters. It requires little konwledge about the results before operating, and the algorithm is flexible to adjust. Standard agglomerative clustering algorithms result in a large number of tiny clusters in a long-tail distribution (shown in Section 6.1). Tiny clusters do not provide sufficient numbers of samples for statistical learning of the refined normal boundary inside each cluster. Standard algorithms also do not handle borderline behaviors, which could be trained in one cluster and tested in another, resulting in false alarms.

Our algorithm (Algorithm 1) clusters program behavior instances based on the co-occurred events in behavior instances. Our algorithm handles the borderline behavior issue – behavior instances on the borderlines are randomly selected into clusters – with a two-step process:

(1) generate scopes of clusters in an agglomerative way (line 11-25)
(2) add behavior instances to generated clusters (line 27-39)

Our algorithm initializes clusters as individual behavior instances (line 2 to line 10), then the nearest clusters are merged (line 11 to line 25) till stopping criteria are reached (line 13) in the first step. The scope of each cluster is calculated in the first step, but not the items in each cluster. Step two fills each cluster with behavior instances and handles borderline behavior (line 27-39).

We use a lazily updated heap h in Algorithm 1 to minimize the calculation and sorting of distances between intermediate clusters. Each entry in h contains the distance between two clusters, and h is sorted based on the distance. The design of the lazily updated heap ensures that a previously merged cluster is not removed proactively in h until the entry containing it is popped and abandoned. Algorithm 1 performs lazy removal of dead clusters in h. Dead clusters refer to the clusters that are merged into others and no longer exist.

The scope of a cluster $C = \{b_i \mid 0 \leq i \leq k\}$ is represented by its event co-occurrence matrix O_C. O_C records occurred events in any behavior instances in C. It is defined in (2) where O_{b_i} is the event co-occurrence matrix of b_i.

$$O_C = O_{b_1} \texttt{ OR } O_{b_2} \texttt{ OR } \ldots \texttt{ OR } O_{b_k},\ 0 \leq i \leq k \tag{2}$$

The distances between *i)* two behavior instances, *ii)* two clusters, and *iii)* a behavior instance and a cluster are all measured by their co-occurrence matrices O_1 and O_2 in (3) where $|O|$ counts the number of `True` in O.

$$dist(O_1, O_2) = \frac{\text{Hamming}(O_1, O_2)}{\min(|O_1|, |O_2|)} \tag{3}$$

ALGORITHM 1: Constrained Agglomerative Program Behavior Clustering

```
Input: a set of normal program behavior instances $B$ and a termination threshold $T_d$.
Output: a set of behavior clusters $\mathbb{C}$.
$h \leftarrow \varnothing_{heap}$; $v \leftarrow \varnothing_{hashtable}$; $V \leftarrow \varnothing_{set}$;
for $b \in B$ do
    $O \leftarrow O_b$;
    $v[O] \leftarrow v[O] + 1$;
    for $O' \in V$ do
        $d_p \leftarrow dist(O, O') \times pen(v[O], v[O'])$;
        push $\langle d_p, O, v[O], O', v[O'] \rangle$ onto $h$;
    end
    add $O$ to $V$;
end
while $h \neq \varnothing_{heap}$ do
    pop $\langle d_p, O_1, v_{O_1}, O_2, v_{O_2} \rangle$ from $h$;
    break if $d_p > T_d$;
    if $O_1 \in V$ and $O_2 \in V$ then
        continue if $v_{O_1} < v[O_1]$ or $v_{O_2} < v[O_2]$;
        $O \leftarrow O_1$ OR $O_2$;
        $v[O] \leftarrow v[O_1] + v[O_2]$;
        remove $O_1$ from $V$; remove $O_2$ from $V$;
        for $O' \in V$ do
            $d_p \leftarrow dist(O, O') \times pen(v[O], v[O'])$;
            push $\langle d_p, O, v[O], O', v[O'] \rangle$ onto $h$;
        end
        add $O$ to $V$;
    end
end
$w[O] \leftarrow \varnothing_{set}$ for all $O \in V$;
for $b \in B$ do
    $O \leftarrow O_b$; $m \leftarrow$ MAXINT;
    for $O' \in V$ do
        if $O$ OR $O' = O'$ then
            if $dist(O, O') < m$ then
                $m \leftarrow dist(O, O')$;
                $V' \leftarrow \{O'\}$;
            else if $dist(O, O') = m$ then
                add $O'$ to $V'$;
            end
        end
    end
    add $b$ to $w[O]$ for all $O \in V'$;
end
$\mathbb{C} \leftarrow \{w[O]$ for all $O \in V\}$;
```

$dist()$: distance function between behaviors/clusters. $pen()$: penalty function for long tail elimination.

Hamming distance alone is insufficient to guide the cluster agglomeration: it loses the semantic meaning of O, and it weighs `True` and `False` the same. However, in co-occurrence matrices, only `True` contributes to the co-occurrence of events.

For example, given four Os

$$O_1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, O_2 = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}, O_3 = \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix}, O_4 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix},$$

O_1 and O_2 are nearer than O_3 and O_4 ($dist(O_1, O_2) = 33\% < dist(O_3, O_4) = 100\%$), while their pure Hamming distances are equal ($\text{Hamming}(O_1, O_2) = \text{Hamming}(O_3, O_4) = 1$).

We explain the unique features of our constrained agglomerative clustering algorithm over the standard design as follows.

- *Long tail elimination:* A standard agglomerative clustering algorithm produces clusters with a long tail distribution of cluster sizes – there are a large number of tiny clusters, and the unbalanced distribution remains at various clustering thresholds. Tiny clusters provide insufficient number of behavior instances to train probabilistic models in INTRA-CLUSTER MODELING.
  In order to eliminate tiny/small clusters in the long tail, our algorithm penalizes $dist(O_1, O_2)$ by (4) before pushing it onto h. $|C_i|$ denotes the size of cluster C_i.
  $$pen(|C_1|, |C_2|) = \max(\log(|C_1|), \log(|C_2|)) \tag{4}$$
- *Penalty maintenance:* The distance penalty between C_1 and C_2 changes when any size of C_1 and C_2 changes. In this case, all entries in h containing a cluster whose size changes should be updated or nullified.
  We use a version control to mark the latest and deprecated versions of clusters in h. The version of a cluster C is recorded as its current size (an integer). It is stored in $v[O]$ where O is the event co-occurrence matrix of C. v is a hashtable that assigns `0` to an entry when the entry is accessed for the first time. A heap entry contains two clusters, their versions and their distance when pushed to h (line 7 and line 21). An entry is abandoned if any of its two clusters are found deprecated at the moment the entry is popped from h (line 15).
- *Borderline behavior treatment:* It may generate a false positive when *i)* $dist(b, C_1) = dist(b, C_2)$, *ii)* b is trained only in C_1 during INTRA-CLUSTER MODELING, and *iii)* a similar behavior instance b' is tested against C_2 in operation OCCURRENCE FREQUENCY ANALYSIS (intra-cluster detection). To treat the borderline behaviors correctly, our clustering algorithm duplicates b in every cluster, which b may belong to (line 27-39). This operation also increases cluster sizes and results in sufficient training in INTRA-CLUSTER MODELING.

4.2 Co-occurrence Analysis (Detection)

This operation performs inter-cluster detection to seek event co-occurrence anomalies. A behavior instance b is tested against all normal clusters $\mathbb{C}$ to check whether the co-occurred events in b are consistent with co-occurred events found *in a single cluster*. An alarm is raised if no such cluster is found. Otherwise, b and its most closely fitted clusters $\mathbb{C}_b = \{C_1, \ldots, C_k\}$ are passed to OCCURRENCE FREQUENCY ANALYSIS for intra-cluster detection.

An incoming behavior instance b fits in a cluster C if $(O_b \texttt{ OR } O_C = O_C) \wedge (O_b \texttt{ AND } O^*_C = O^*_C)$ where O_C and O_b are the event co-occurrence matrices of C and b, and O^*_C is a feature of C defined in (5). We define the fitting test to fulfill the event co-occurrence analysis and detects anomalous behaviors with fabricated or broken co-occurrences.

$$O^*_C = O_{b_1} \texttt{ AND } O_{b_2} \texttt{ AND } \ldots \texttt{ AND } O_{b_k}, \; 0 \leq i \leq k \tag{5}$$

The detection process searches for all clusters in which b fits, i.e., $\mathbb{C}_b$. If $\mathbb{C}_b \neq \varnothing$, distances between b and each cluster in $\mathbb{C}_b$ are calculated using (3). The clusters with the nearest distance ($|\mathbb{C}_b| \geq 1$) are selected as $\mathbb{C}_b$.

4.3 Intra-cluster Modeling (Training)

Within a cluster C, LAD analyzes behavior instances through their transition frequency matrices $\{F_b \mid b \in C\}$. The matrices are vectorized into data points in a high-dimensional detection space where each dimension records the occurrence frequency of a specific event across profiles. Two analysis methods reveal relations among frequencies.

The probabilistic method. We employ a one-class SVM, i.e., ν-SVM [48], to seek a frontier $\mathcal{F}$ that envelops all behavior instances $\{b \mid b \in C\}$.

a) Each frequency value is preprocessed with a logarithmic function $f(x) = \log_2(x + 1)$ to reduce the variance between extreme values (empirically proved necessary).
b) A subset of dimensions are selected through *frequency variance analysis* (FVA)[6] or *principle component analysis* (PCA)[7] [43] before data points are consumed by ν-SVM. This step manages the *curse of dimensionality*, a common concern in high-dimensional statistical learning.
c) We pair the ν-SVM with a kernel function, i.e., radial basis function (RBF)[8], to search for a non-linearly $\mathcal{F}$ that envelops $\{b \mid b \in C\}$ tightly. The kernel function transforms a non-linear separating problem into a linearly separable problem in a high-dimensional space.

The deterministic method. We employ *variable range analysis* to measure frequencies of events with zero or near zero variances across all program behaviors $\{b \mid b \in C\}$.

Frequencies are discrete integers. If all frequencies of an event in different behavior instances are the same, PCA simply drops the corresponding dimension. In some clusters, all behavior instances (across all dimensions) in C are the same or almost the same. Duplicated data points are treated as a single point, and they cannot provide sufficient information to train probabilistic models, e.g., one-class SVM.

Therefore, we extract deterministic rules for events with zero or near zero variances. This model identifies the frequency range $[f_{min}, f_{max}]$ for each of such events. f_{min} can equal to f_{max}.

4.4 Occurrence Frequency Analysis (Detection)

This operation performs intra-cluster detection to seek quantitative frequency relational anomalies: *i)* deviant relations among multiple event occurrence frequencies, and/or *ii)* aberrant occurrence frequencies. Given a program behavior instance b and its closely fitted clusters $\mathbb{C}_b = \{C_1, \ldots, C_k\}$ discovered in Co-occurrence Analysis, this operation tests b in every C_i $(0 \leq i \leq k)$ and aggregates the results using (6).

$$\exists C \in \mathbb{C}\ N_{clt}(b, C) \Rightarrow b \text{ is normal} \tag{6}$$

The detection inside C is performed with three rules, and the result is aggregated into $N_{clt}(b, C)$ through logical conjunction of available rules[9]:

$$N_{clt}(b, C) \leftarrow \texttt{True} \text{ iff } b \text{ is tested normal by all applicable rules to } C.$$

- *Rule 1: normal if the behavior instance b passes the probabilistic model detection.* The frequency transition matrix F of b is vectorized into a high-dimensional data point and tested against the one-class SVM model built in Intra-cluster Modeling. This operation computes the distance d between b and the frontier $\mathcal{F}$ established in the ν-SVM. If b is within the frontier or b is on

[6]FVA selects dimensions/events with larger-than-threshold frequency variances across all behavior instances in C.
[7]PCA selects linear combinations of dimensions/events with larger-than-threshold frequency variances, which is a generalization of FVA.
[8]Multiple functions have been tested for selection.
[9]Not all three rules could be applicable to an arbitrary cluster C.

the same side as normal behavior instances, then $d > 0$. Otherwise, $d < 0$. d is compared with a detection threshold T_f that $T_f \in (-\infty, +\infty)$. b is abnormal if $d < T_f$.

- *Rule 2: normal if the behavior instance b passes the deterministic model detection.* Events in b with zero or near zero variances are tested against the deterministic model built in Intra-cluster Modeling. b is abnormal if any event frequency of b exceeds its normal range.
- *Rule 3: presumption of innocence in tiny clusters.* If no frequency model is trained in C because the size of C is too small, the behavior instance b is marked as normal. This rule sacrifices the detection rate for reducing false alarms in insufficiently trained clusters.

5 IMPLEMENTATION

We implement a prototype of LAD on Linux (Fedora 21, kernel 3.19.3). The static analysis is realized through `C` (`ParseAPI` [42]). The profiling, training, and detection phases are realized in `Python`. The dynamic tracing and behavior recognition are realized through Intel `Pin`, a leading dynamic binary instrumentation framework, and `SystemTap`, a low-overhead dynamic instrumentation framework for Linux kernel. Tracing mechanisms are independent of our detection design; more efficient tracing techniques can be plugged in replacing `Pin` and `SystemTap` to improve the overall performance in the future.

Static analysis before profiling: symbols and address ranges of routines/functions are discovered for programs and libraries. The information helps to identify routine symbols if not found explicitly in dynamic tracing. Moreover, we leverage static analysis to list legal caller-callee pairs.

Profiling: Our prototype *i)* verifies the legality of events (function calls) in a behavior instance b and *ii)* profiles b into two matrices (Section 3.1). The event verification filters out simple attacks that violate control flows before our approach detects stealthy aberrant path attacks. We implement profile matrices in Dictionary of Keys (DOK) format to minimize storage space for sparse matrices.

Dynamic tracing and behavior recognition: We develop a Pintool in JIT mode to trace function calls in the user space and to recognize boundaries of long trace segments within entire program executions. Our Pintool is capable of tracing *i)* native function calls, *ii)* library calls *iii)* function calls inside dynamic libraries, *iv)* kernel thread creation and termination. Traces of different threads are isolated and stored separately. Our Pintool recognizes whether a call is made within a given routine and on which nested layer the given routine executes (if nested execution of the given routine occurs). This functionality enables the recognition of long trace segments through routine boundary partitioning.

We demonstrate that our approach is versatile recognizing program behaviors at different granularities. We develop a `SystemTap` script to trace system calls with timestamps. It enables long trace partitioning via activity intervals when the program is monitored as a black box.

6 EVALUATIONS

We extensively evaluate the detection accuracy, capability and performance of our LAD system. We aim to answer the following questions in this section:

1. How well does our approach detect real-world aberrant path attacks? (Section 6.2)
2. How accurate is our approach recognizing anomalous program behavior instances? (Section 6.3)
3. What's the advantage of our constrained agglomerative clustering algorithm? (Section 6.4)
4. How much overhead does our detection incur? (Section 6.5)

6.1 Experiment Setup

We study three programs/libraries (Table 2) in distinct categories. We demonstrate that LAD is a versatile detection solution to be applied to programs and dynamic libraries with various event

Table 2. The profile information of programs/libraries and statistics of normal profiles

		Profile Overview			Average Normal Profile		
Program	**Version**	**Event Set**	**Segmentation**	**#(N.P.)**	**#(Sym.)**	**#(Event)**	**#(U.E.)**
`sshd`	1.2.30	function calls	routine boundary	4800	415	34511	180
`libpcre`	8.32	function calls	library call	17360	79	44893	45
`sendmail`	8.14.7	system calls†	event statistics	6579	350	1134	213

N.P., Sym., and U.E. are short for *normal profile*, *symbol*, and *unique event*, respectively.
An event is a function or system call (detailed in column Event Set).
A normal profile is recorded in a matrix O and a matrix F.
A symbol defines a caller or a callee of an event.
†Function calls are not traced due to its complex process spawning logic. It requires customization of our Pintool to trace them.

	`sshd`	`libpcre`	`sendmail`
T_d	1.6	2.0	1.6
$\lvert\mathbb{C}\rvert$	6	29	20

Table 3. Overview of program behavior clustering.

definitions and long trace segmentation means. We detail the programs/libraries and their training dataset (normal profiles) below.

`[sshd]` Long trace segment definition: all function calls within routine `do_authentication()` of `sshd` (SLOC = 19,215)[10]. The routine `do_authentication()` is called in a forked thread after a client initializes its connection to `sshd`. All session activities occur within the long trace segment. Normal runs cover three authentication methods (password, public key, rhost), each of which contains 800 successful and 800 failed connections. 128 random commands are executed in each successful connection.

`[libpcre]` Long trace segment definition: all function calls inside `libpcre` when a library call is made and control flows go into `libpcre` (SLOC = 68,017). Library calls are triggered through `grep -P`. Over 10,000 normal tests are used from the `libpcre` package.

`[sendmail]` Long trace segment definition: a continuous system call sequence wrapped by long no-op (no system call) intervals. `sendmail` is an event-driven program that only emits system calls when sending/receiving emails or performing a periodical check. We set up this configuration to demonstrate that LAD can consume various events, e.g., system calls. We collect over 6,000 normal profiles on a public `sendmail` server during 8 hours.

We list clustering threshold T_d used for the three studied programs/libraries in Figure 3[11]. $|\mathbb{C}|$ denotes the number of clusters computed with the specific T_d.

In operation Occurrence Frequency Analysis, the detection threshold T_f is determined by a given false positive rate (FPR) upper bound, i.e., FPR^u, through cross-validation. In the training

[10]The actually monitored part accessible from `do_authentication()` is smaller than the entire program.
[11]The value is empirically chosen to keep a balance between an effective recognition of diverse behaviors and an adequate elimination of tiny clusters.

Table 4. Overview of reproduced attacks and detection results

Attack	Target	Attack Settings	#(A.)	D.R.	FPR^u
FVO[a]	`sshd`	an inline virtual exploit that matches a username	800	100%	0.0001
ReDoS	`libpcre`	3 deleterious patterns paired with 8-23 input strings	46	100%	0.0001
DHA[b]	`sendmail`	probing batch sizes: 8, 16, 32, 64, 100, 200, and 400	14	100%	0.0001

[a] flag variable overwritten attack.
[b] directory harvest attack.
*Note:*A. is short for *attack attempt*. D.R. is short for *detection rate*. FPR^u is the false positive rate upper bound (details in Section 6.1).

phase of cross-validation, we perform multiple random 10-fold partitioning. Among distances from all training partitions, T_f is initialized as the kth smallest distance within distances[12] between a behavior instance and the ν-SVM frontier $\mathcal{F}$. k is calculated using FPR^u and the overall number of training cases. The FPR is calculated in the detection phase of cross-validation. If $\text{FPR} > \text{FPR}^u$, a smaller k is selected until $\text{FPR} \leq \text{FPR}^u$.

6.2 Detecting Real-World Attacks

We reproduce three known aberrant path attacks to test the detection capability of our LAD system. Our prototype detects all attack attempts, and the detection overview is in Table 4.

Flag Variable Overwritten Attack (FVO in Table 4) is a non-control data attack. An attacker tampers with decision-making variables. The exploit takes effect when the manipulated data affects the control flow at some later point of execution.

We reproduce the flag variable overwritten attack against `sshd` introduced by Chen et al. [6]. We describe the attack in Section 2.1, bullet (a) and in Figure 1. We simplify the attack procedure by placing an *inline virtual exploit* in `sshd` right after the vulnerable routine `packet_read()`:

```
if (user[0] == 'e' && user[1] == 'v' && user[2] == 'e') authenticated = 1;
```

This inline virtual exploit produces the immediate consequence of a real exploit – overwriting `authenticated`. It does not interfere with our tracing/detection because no `call` instruction is employed. Each attack attempt is constructed with the user name "eve" and 128 random shell commands after a successful login.

Our approach (configured at FPR^u 0.0001) successfully detects all attack attempts in inter-cluster detection (Co-occurrence Analysis)[13]. We present normal and attack traces inside long trace segments (selected routine `do_authentication()`) in Figure 5 to illustrate the detection.

In Figure 5, the *Attack* and *Normal*[b] bear the same trace prior to the last line, and the *Attack* and *Normal*[a] bear the same trace after (including) the last line. Our approach detects the attack with event co-occurrence analysis: the control-flow segment containing `do_auth > debug` should not co-occur with the control-flow segment containing `do_auth > do_authed` (and following calls) in one long trace segment.

In the traces, there are identical 218 `call` events including library routines (36 `calls` excluding library ones) between the third line and the last line in Figure 5. We test an n-gram detection tool, and it requires at least $n = 37$ to detect the specific attack without libraries routine traced. The 37-gram model results in an FPR of 6.47% (the FPR of our approach is less than 0.01%). This indicates

[12] The distance can be positive or negative. More details are specified in Rule 1 (Section 4.4).
[13] One-class SVM in Occurrence Frequency Analysis only detects 3.8% attack attempts if used alone.

Table 5. Samples of normal and anomalous sshd traces

Normal (successfully authenticated)	**Normal** (failed authentication)	**Attack** (flag variable overwritten attack)
...	...	...
auth_p > xfree	auth_p > xfree	auth_p > xfree
do_auth > xfree	do_auth > debug	do_auth > debug
do_auth > log_msg	do_auth > xfree	do_auth > xfree
do_auth > p_start	do_auth > p_start	do_auth > p_start
p_start > buf_clr	p_start > buf_clr	p_start > buf_clr
...	...	...
phdtw > buf_len	phdtw > buf_len	phdtw > buf_len
do_auth > do_autd	do_auth > p_read	do_auth > do_autd
...	...	...

Note:"caller > callee" denotes a function call. Routine names are abbreviated to save space.

Table 6. Deleterious patterns used in ReDoS attacks

	Deleterious Pattern	**#(Attack Attempts)**
Pattern 1	`^(a+)+$`	15
Pattern 2	`((a+))+$`	8
Pattern 3	`^(([a-z])+.)+[A-Z]([a-z])+$`	23

that n-gram models with a large n is difficult to converge at training. We do not test automaton-based detection that defines illegal transitions as anomalous behaviors. Automaton-based methods cannot detect the attack in theory since there are no illegal function calls.

Flag Variable Overwritten Attack (ReDoS in Table 4) is a service abuse attack. It exploits the exponential time complexity of a regex engine when performing backtracking. The attacks construct extreme matching cases where backtracking is involved. All executed control flows are legal. The regex engine hangs due to the extreme complexity.

We produce 46 ReDoS attack attempts targeting `libpcre`[14]. Three deleterious patterns are used (Table 6). For each deleterious pattern, attacks are constructed with an increasing length of `a` in the input string starting at 6, e.g., `aaaaaaaab`. We stop attacking `libpcre` at different input string lengths so that the longest hanging time periods for different deleterious patterns are about the same (a few seconds). A longer input string incurs a longer hanging time; it results in a more severe ReDoS attack than a shorter one.

ReDoS attacks are detected in intra-cluster detection operation (Occurrence Frequency Analysis) by the probabilistic method, i.e., ν-SVM. We test our LAD system with both PCA and FVA feature selection (Section 4.3, the probabilistic method, bullet b). The detection results (Figure 4) show that LAD configured with PCA is more sensitive than it configured with FVA. LAD (with PCA) detects all attack attempts at different FPRs[15]. The undetected attack attempts (with FVA) are all constructed with the small amount of `a` in the input strings, which do not result in very severe ReDoS attacks.

[14]Internal deep recursion prevention of `libcpre` is disabled.
[15]No attack is detected if only Co-occurrence Analysis is performed.

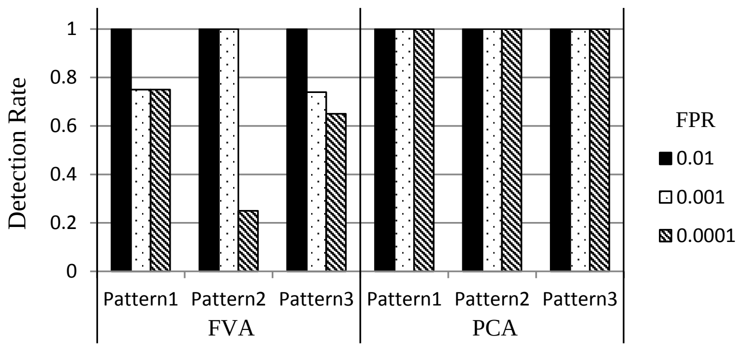


Fig. 4. Detection rates of ReDoS attacks.

Directory Harvest Attack (DHA in Table 4) is a service abuse attack. It probes valid email users through brute force. We produce 14 DHA attack attempts targeting `sendmail`. Each attack attempt consists of a batch of closely sent probing emails with a dictionary of possible receivers. We conduct DHA attacks with 7 probing batch sizes from 8 to 400 (Table 4). Two attack attempts are conducted for each batch size.

Our approach (configured at FPR^u 0.0001) successfully detects all attack attempts with either PCA or FVA feature selection[15]. DHA attacks are detected in intra-cluster detection (Occurrence Frequency Analysis) by the probabilistic method, i.e., ν-SVM. The attacks bypass the inter-cluster detection (Co-occurrence Analysis) because invalid usernames occur in normal training dataset.

This experiment demonstrates that our LAD system can consume coarse program behavior descriptions (e.g., system calls) to detect attacks. Most of the probing emails do not have valid receivers. They result in a different processing procedure than that for normal emails; the batch of DHA emails processed in a long trace segment gives anomalous ratios between frequencies of *valid email processing control flows* and frequencies of *invalid email processing control flows*. In `sendmail`, these different control flows contain different sets of system calls, so they are revealed by system call profiles. More precise detection requires the exposure of internal program activities, such as call, jump and return instructions in the program user space.

6.3 Systematic Accuracy Evaluation

We systematically demonstrate how sensitive and accurate our approach is through receiver operating characteristic (ROC). Besides normal program behaviors ground truth (Section 6.1), we generate four types of synthetic aberrant path anomalies. We first construct F' for each synthetic anomalous behavior instance b', and then we use (1) to derive O' (of b') from F'.

1. *Montage anomaly*: two behavior instance b_1 and b_2 are randomly selected from two different behavior clusters. For a cell $f'_{i,j}$ in F', if one of $f_{1_{i,j}}$ (of F_1) and $f_{2_{i,j}}$ (of F_2) is 0, the value of the other is copied into $f'_{i,j}$. Otherwise, one of them is randomly selected and copied.
2. *Incomplete path anomaly*: random one-eighth of non-zero cells of a normal F are dropped to 0 (indicating events that have not occurred) to construct F'.
3. *High-frequency anomaly*: three cells in a normal F are randomly selected, and their values are magnified 100 times to construct F'.
4. *Low-frequency anomaly*: similar to high-frequency anomalies, but the values of the three cells are reduced to 1.

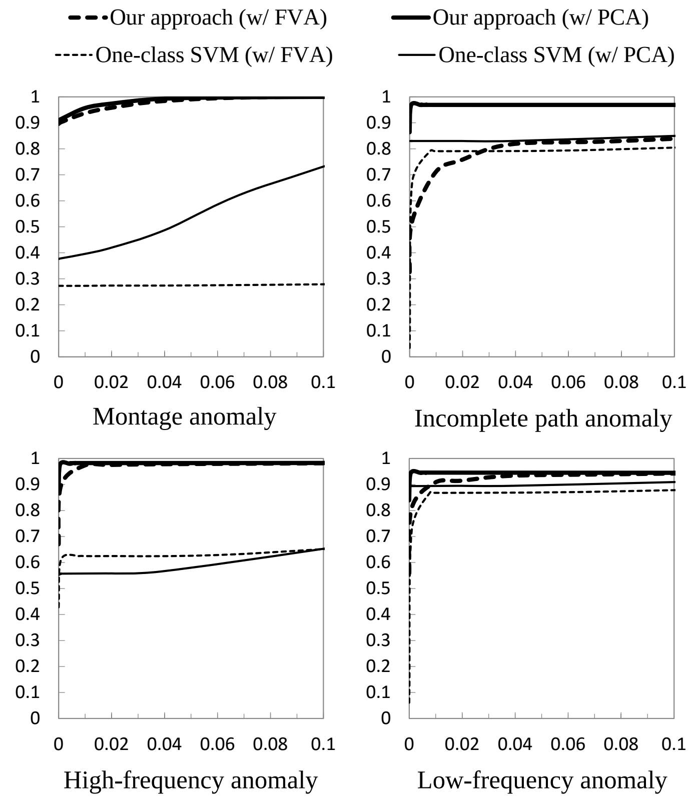


Fig. 5. `libpcre` ROC of LAD (our approach) vs. basic one-class SVM. X-axis is false positive rate, and y-axis is detection rate.

Table 7. Semantics of true/false positives/negatives

	Abnormal Behavior	**Normal Behavior**
Anomaly detected	*true positive*	*false positive*
No anomaly detected	*false negative*	*true negative*

To demonstrate the effectiveness of our design in handling diverse program behaviors, we compare our LAD system with a basic one-class SVM (the same ν-SVM and same configurations, e.g., kernel function, feature selection, and parameters, as used in the Intra-cluster Modeling operation of LAD).

We present the detection accuracy results on `libpcre` in Figure 5, which has the most complicated behavior patterns among the three studied programs/libraries[16]. In any subfigure of Figure 5, each dot is associated with a false positive rate (multi-round 10-fold cross-validation with 10,000 test cases) and a detection rate (1,000 synthetic anomalies). We define positives and negatives as in Table 7 – an anomaly is a positive.

[16]Results of the other two programs share similar characteristics as `libpcre` and are not presented.

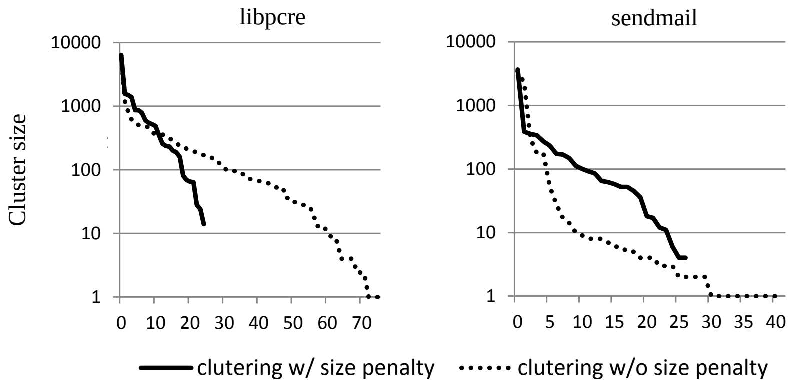


Fig. 6. Long-tail elimination with our constrained agglomerative clustering algorithm.

Figure 5 shows the effectiveness of our clustering design. The detection rate of our prototype (with PCA[17]) is usually higher than 0.9 with FPR less than 0.01. Because of diverse patterns, basic one-class SVM fails to learn tight boundaries that wrap diverse normal patterns as expected. A loose boundary results in false negatives and low detection rates.

6.4 Constrained Agglomerative Clustering Algorithm

We illustrate the effectiveness of our constrained agglomerative clustering algorithm design to eliminate long tails of cluster size distributions. The optimization supports adequate training in each cluster and minimizes the use of the *innocence rule* in Co-occurrence Analysis (Section 4.4). The rule may result in false negatives.

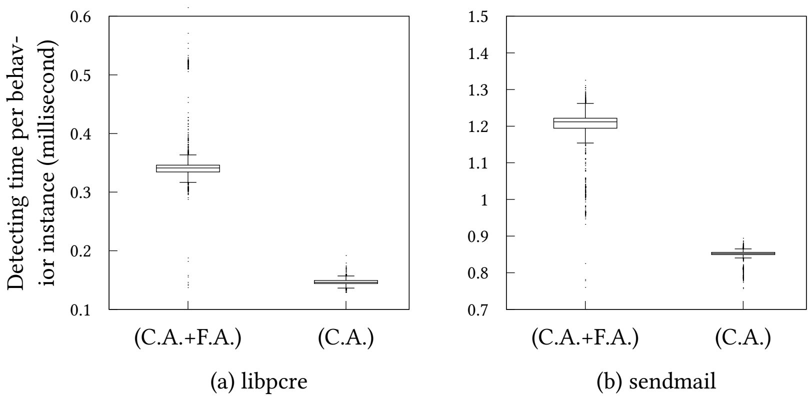
Figure 6 compares the clustering results between standard agglomerative clustering algorithm and our constrained design regarding cluster size distribution: the dotted line denotes the results of the standard agglomerative clustering algorithm as the baseline. The solid line denotes the results of the proposed constrained algorithm with size penalty. Our design eliminates tiny clusters in the long tail of cluster size distributions. Tiny clusters contain insufficient numbers of behavior instances for building probabilistic models in Intra-cluster Modeling. The constrained algorithm prioritizes the merge of tiny clusters into bigger ones by giving a cluster size penalty when calculating the distances between clusters (line 20 in Algorithm 1).

6.5 Performance Analysis

A fast runtime detection is important for enabling real-time protection and minimizing negative user experience [39]. The overall overhead of a program anomaly detection system comes from *tracing*, *training* and *analysis* in general.

We evaluate the performance of LAD analysis procedures (inter- and intra-cluster detections) with either function call profiles (`libpcre`) or system call profiles (`sendmail`). We test the analysis on all normal profiles (`libpcre`: 17360, `sendmail`: 6579) to collect overhead for *inter-cluster detection alone* and *the combination of inter- and intra-cluster detection*[18]. The analysis of each behavior instance is repeated 1,000 times to obtain a fair timing. The performance results in Figure 7 illustrate that

[17]PCA proves itself more accurate than FVA in Figure 5.
[18]PCA is used for feature selection. FVA (results omitted) yields a lower overhead due to its simplicity.



Note: C.A.+F.A.: Inter- and intra-cluster detection combined.
C.A.: Inter-cluster detection only.

Fig. 7. Detection (analysis) overhead of our LAD system.

- It takes 0.1~1.3 ms to analyze a single behavior instance, which contains 44893 function calls (`libpcre`) or 1134 system calls (`sendmail`) on average (Table 2).
- The analysis overhead is positively correlated with the number of unique events in a profile (Table 2), which is due to our DOK implementation of profile matrices.
- Anomalies detected at Co-occurrence Analysis takes less time to detect than anomalies detected at Occurrence Frequency Analysis because only the first stage is involved in the detection.

Two major factors affect the detection time on a program behavior instance: *i)* the symbol set size, which determines the size of a profile matrix; and *ii)* the number of unique events in a profile, which determines the number of non-zero entries in a matrix. Our prototype is implemented using dictionary of keys (DOK), thus the detection performance of our prototype should be affected more by the number of unique events for different programs (the last column in Table 2).

Compared with the analysis procedure, dynamic function call tracing incurs a noticeable overhead. `sshd` experiences a 167% overhead on average when our Pintool is loaded. A similar 141% overhead is reported by Jalan and Kejariwal in their dynamic call graph Pintool `Trin-Trin` [31]. Advanced tracing techniques, e.g., Intel Processor Tracing (PT) [33], combined with the latest hardware, e.g., Skylake architecture, can potentially reduce the tracing overhead to less than 5% toward a real-time detection system [16].

Another choice to deploy LAD is to profile program behaviors through system calls as we demonstrate using `sendmail`. System calls can be traced through `SystemTap` with near-zero overhead [57], but it sacrifices the capability to reveal user-space program activities and downgrades the modeling/detection accuracy.

LAD supports offline detection or forensics of program attacks, in which case accuracy is the main concern instead of performance [56]. Our Pintool enables analysts to locate anomalies within long program traces, and our matrices provide caller information for individual function calls. This information helps analysts quickly reduce false alarms and locate vulnerable code segments. For potential online detection deployment, tracing is the bottleneck as shown above. Finding a faster way to monitor the system, e.g., Intel PT, gives more boost to the entire procedure than detection procedure improvements. A comprehensive benchmark is also needed to evaluate the overall overhead of an online detection system in terms of different tracing mechanisms, different styles of programs, and different numbers of hooks for different monitoring needs.

Although training performance is not a typical concern for program anomaly detection, the procedure should be practical to enable the use of the approach. Our prototype shows reasonable training time from 1 to 100 seconds in all three cases we tested. The more complicated and diverse the behaviors are, the longer the algorithm takes to figure out the clusters. System calls in the `sendmail` case characterize full-program behaviors in 8 hours, which constructs the most diverse dataset. Over 6000 behavior instances take 86.17s to train on average. Behaviors in `libpcre` and `sshd` are defined in a more focused manner, e.g., activities within a routine or library call. They take 5.01s and 1.77s to train 17360 and 4800 behavior instances on average.

Summary We evaluate the detection capability, accuracy, and performance of our detection prototype on Linux.

- Our approach successfully detects all reproduced aberrant path attack attempts against `sshd`, `libpcre` and `sendmail` with less than 0.0001 false positive rates.
- Our approach is accurate in detecting different types of synthetic aberrant path anomalies with a high detection rate (> 0.9) and a low false positive rate (< 0.01).
- Our approach analyzes program behaviors fast; it only incurs 0.1~1.3 ms analysis overhead (excluding tracing) per behavior instance (1k to 50k function/system calls in our experiments).

7 DISCUSSIONS

This paper aims to present a practical program anomaly detection approach at the context-sensitive language level. It does not accomplish all context-sensitive features, e.g., order of events, yet it correlates events far away and sheds light on the development towards more advanced context-sensitive detection models.

Prior to and after the presented approach, several steps can be made to complete the detection, ease the adoption, and reduce potential false positives. We first discuss potential improvements from the deployment perspective below, and then we discuss the accuracy issues as well as potential improvements.

- *Fine-grained tracing with low overhead.* As shown in Section 6.5, the detection of a single behavior instance (>10k events) only takes 0.1~1.3 ms. What impedes the approach to be used as an online system is the tracing overhead. Practical fine-grained user-space tracing is the bottleneck of many program anomaly detection systems based on dynamic analysis. Fast and practical tracing mechanisms, e.g., Intel PT (discussed in Section 6.5), could open the door to online detection leveraging the proposed approach.
- *Long trace segment definition.* The proposed approach recognizes long trace segments based on the definitions specified by security analysts or software developers. In our evaluation, we use three simple long trace segment definitions: *i)* subroutine boundary for `sshd`, *ii)* library call boundary for `libpcre`, and *iii)* statistical event density for `sendmail`. The third is the default choice if no specific knowledge is revealed about the program or attacks, yet it is the coarsest. This ad-hoc procedure can be improved by developing a segment definition construction algorithm. The algorithm either takes in vanilla programs plus external knowledge or compiled programs with security tags specified by developers. It could list and rank generated long segment definitions to aid the deployment and configuration of the proposed approach.

- *False alarms with incomplete training.* We recommend deploying the proposed system for monitoring server programs or applications with relatively static behaviors. Complicated user applications such as browsers have a vast behavior space, which is difficult to cover by limited training samples. False positives could be generated if normal behaviors are not present or normal boundaries do not reach far enough to cover the tested behavior instances.
  Our model discards the order information between events, which projects the original behavior space onto a lower dimensional subspace for practical modeling. However, incomplete training is still an issue for programs with dynamic behavior patterns. Further reducing the space can result in the loss of context-sensitive information and leads to less precise context-free language models like DFA models.

Since the proposed approach sacrifices precision for practicality in its design, potential mimicry attacks could be constructed to exploit the gap between $\tilde{L}$ and the most precise program execution description. Additionally, the learning-detection cycle may pose potential inaccuracy to detection results. We discuss attacks that can potentially evade the detection of our model as well as possible countermeasures.

- *Exploiting the granularity of our model.* An attacker may construct an attack: *i)* it consists of control-flow segments without `call` instructions, and *ii)* it does not incur future anomalous control flows containing `call` after exploit, e.g., only substituting arguments of legitimate `exec` system call to invoke `/bin/sh`. Our current prototype does not detect such a threat. However, this is not a theoretical defeat. The prototype can be extended to monitor and correlate non-`call` instructions and subprocess behaviors for detecting this attack.
- *Exploiting the orderless characteristic of our model.* Our approach does not enforce the order of events within a program behavior instance. The property could be exploited by an attacker. Directly adding event order information on top of our model results in upgrading $\tilde{L}$ to a linear bounded automaton (LBA), which incurs *exponential training convergence time* discussed in Section 2.3. It remains an open question to construct LBAs, probably with heuristics, for program anomaly detection.
- *Exploiting the normal boundary of our detection.* We employ a stochastic language in our approach – the one-class SVM estimates a proper boundary of normal behaviors in each of the clusters – to mitigates false positives due to insufficient training samples. The estimation opens potential opportunities for attackers to escape the detection, i.e., false negatives.
  First, an attacker can sacrifice the effectiveness of attack for not being detected. For example, in a mild brute force SSH password attack, low frequency of attack attempts makes it unlikely to be discovered. The parameter FPR^u in our prototype establishes a balance between missing attacks and oversensitive detection that yields false alarms. We provide a quantitative analysis in Section 6.3 to understand the sensitivity of our prototype and this can be further improved by choosing other boundary computation methods and parameters.
  Second, if an attacker has the privilege to reach the training phase, one can potentially poison the model and bias the system towards incorrect detection results, e.g., causative attacks where the attacker influences the training data [41]. In theory, our detection approach is vulnerable to this attack due to the use of a stochastic language. However, our approach is not designed to be an online machine learning system – the model is not constantly updated – and it avoids the problem to some extent. In general, data sanitization provides a more precise model and reduce false negatives for any learning system. Moreover, exploration in adversarial machine learning [26] can further enable the development of online learning algorithms for program anomaly detection in the future.

8 RELATED WORK

Conventional program anomaly detection systems (a.k.a. host-based intrusion detection systems) follow Denning's intrusion detection vision [8]. They were designed to detect illegal control flows or anomalous system calls based on two basic paradigms: *i)* n-gram short call sequence validation that was introduced by Forrest et al. [12]; and *ii)* automaton transition verification, which was first described by Kosoresow and Hofmeyr [34] (DFA) and formalized by Sekar et al. [49] (FSA) and Wagner and Dean [59] (NDPDA).

Each path leads to a fruitful line of models for detecting anomalous program behaviors. The basic n-gram model was further studied in [11, 28] and sophisticated forms of it were developed, e.g., machine learning models [29, 36], first-order Markov models [64, 65], hidden Markov models [15, 61], and neural network models [17]. Call arguments were used to precisely define states in [38]. n-gram frequencies were studied in [27]. Beyond program anomaly detection, n-grams was also used in malware detection [5].

The essence of n-gram is to model and analyze local features of program traces with a small n. Enlarging n results in exponential convergence and storage issues [11]. However, small n (local feature analysis) makes it possible for attackers to evade the detection by constructing a malicious trace, of which any small fragment is normal. Wagner and Soto first demonstrated such a mimicry attack with a malicious sequence of system calls diluted to normal [60].

The other path, i.e., automaton detection, aims to model a program not restricted to short call sequence analysis. It builds an automaton or a pushdown automaton to read the entire trace at a time. However, reading the entire trace is not equivalent to correlating events in the transition history. All automaton models in literature are first-order and they only verify each state transition on its own. Program counter and call stack information were used to help precisely define each state, e.g., a system call, in an automaton [9, 10, 49]. Static analysis was used [59, 63] and Pushdown automaton or its equivalents were employed in many advanced models [9, 19, 30, 37]. Hidden procedure transition information is revealed by inserting flags in particular procedures [19]. Call arguments were added in FSA models [18] to improve modeling accuracy, and individual transition frequencies had been analyzed to detect DoS attacks [14].

Some of the existing detection methods are *context sensitive* [9, 10, 19, 20, 51, 62], but they are not context-sensitive language models. These models use calling context information to help identify a program state, e.g., a system call. It is different from event co-occurrence analysis because the call stack only provides details at a stack snapshot. It is not designed to record all historical execution paths. Events are forgotten when their associated calls are popped from the stack, and they cannot be observed by later events. Event co-occurrence analysis is also not equivalent to *path sensitivity* since it does not require the analysis of the order of events.

The relation among events that occur far away has not been systematically studied in the literature. The missing analysis allows aberrant path attacks to take place. In this paper, we formalize the problem of event correlation analysis within long program traces using a stochastic mildly context-sensitive language and bring forward a detection system that correlates events in long trace segments for event co-occurrence analysis and occurrence frequency analysis. Frequency analysis has been made as extensions to existing models [14, 27], but they are restricted by the underlying automaton/n-gram models (regular or context-free languages).

Clustering and classification techniques have been widely used in malware classification [1, 13, 32, 46]. Malware classification aims at extracting abstract malware behavior *signatures* and identifies a piece of malware using one or multiple signatures. However, program anomaly detection models normal behaviors and exams an *entire* profile to decide whether it is normal. It is not sufficient to conclude an incoming behavior is normal that one feature of it is normal.

Correlation analysis techniques were developed to detect network intrusions. Valeur et al. described a comprehensive framework to correlate alerts from various IDS systems [58]. Perdisci et al. proposed 2_v-gram scheme to discover related bytes v positions apart in traffic payload [44, 45]. Correlation analysis, including self-correlation, was used for detecting synchronized and repetitive bot activities [21]. Zhang et al. proposed a system to infer network traffic correlation using pre-defined rules [66] and machine learning methods [67]. These techniques were developed under specific bot behavior hypotheses, e.g., temporal relations of network events. In comparison, we address the program anomaly detection problem by developing new algorithms to overcome the unique behavior diversity and scalability challenges.

Defenses against specific known program attacks have been investigated besides anomaly detection. For example, Moore et al. introduced *backscatter analysis* to discover DoS attacks [40], and Brumley et al. invented RICH to prevent integer overflow [4]. These defenses target specific attack signatures and cannot detect unknown attacks. Therefore, they are different from general anomaly detection approaches.

9 CONCLUSIONS AND FUTURE WORK

In this paper, we studied aberrant path attacks, formalized a stochastic mildly context-sensitive language anomaly detection model, and presented a two-stage data mining approach to detect these attacks in long program traces. Our work points out the need for a context-sensitive language level model in response to the development of modern attacks where attackers alter execution paths but not the control flows. Our work demonstrates the effectiveness of large-scale program behavior modeling via event correlation analysis in long program traces and sheds light on the future development of comprehensive context-sensitive language detection model. In future work, we plan to adopt advanced dynamic tracing techniques and build real-time security incidence response systems to enforce program execution security on top of our detection solution.

REFERENCES

[1] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Krügel, and Engin Kirda. 2009. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, Reston, VA, USA, 8–11.

[2] Johan Behrenfeldt. 2009. *A Linguist's Survey of Pumping Lemmata*. Master's thesis. University of Gothenburg.

[3] S. Bhatkar, A. Chaturvedi, and R. Sekar. 2006. Dataflow anomaly detection. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 15–62.

[4] David Brumley, Dawn Xiaodong Song, Tzi cker Chiueh, Rob Johnson, and Huijia Lin. 2007. RICH: Automatically Protecting Against Integer-Based Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, Reston, VA, USA, 1–28.

[5] Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. 2012. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 122–132.

[6] Shuo Chen, Jun Xu, Emre C Sezer, Prachi Gauriar, and Ravishankar K Iyer. 2005. Non-control-data attacks are realistic threats. In *Proceedings of the USENIX Security Symposium*, Vol. 14. USENIX Association, Berkeley, CA, USA, 12–12.

[7] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. 2007. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*. Springer, Berlin, Germany, 63–86.

[8] Dorothy E Denning. 1987. An intrusion-detection model. *IEEE Transactions on Software Engineering* 13, 2 (1987), 222–232.

[9] Henry Hanping Feng, Jonathon T Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P Miller. 2004. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 194–208.

[10] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. 2003. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society,

Washington, DC, USA, 62–75.

[11] Stephanie Forrest, Steven Hofmeyr, and Anil Somayaji. 2008. The evolution of system-call monitoring. In *Proceedings of the Annual Computer Security Applications Conference*. IEEE Computer Society, Washington, DC, USA, 418–430.

[12] Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. 1996. A sense of self for Unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 120–128.

[13] Matt Fredrikson, Somesh Jha, Mihai Christodorescu, Reiner Sailer, and Xifeng Yan. 2010. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 45–60.

[14] Alessandro Frossi, Federico Maggi, Gian Luigi Rizzo, and Stefano Zanero. 2009. Selecting and improving system call models for anomaly detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, Hamburg, Germany, 206–223.

[15] Debin Gao, Michael K Reiter, and Dawn Song. 2006. Behavioral distance measurement using hidden Markov models. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*. Springer, Hamburg, Germany, 19–40.

[16] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 585–598.

[17] Anup K Ghosh, James Wanken, and Frank Charron. 1998. Detecting anomalous and unknown intrusions against programs. In *Proceedings of the 14th Annual Computer Security Applications Conference*. IEEE Computer Society, Washington, DC, USA, 259–267.

[18] Jonathon T Giffin, David Dagon, Somesh Jha, Wenke Lee, and Barton P Miller. 2006. Environment-sensitive intrusion detection. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*. Springer, Hamburg, Germany, 185–206.

[19] Jonathon T Giffin, Somesh Jha, and Barton P Miller. 2004. Efficient Context-Sensitive Intrusion Detection. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, Reston, VA, USA, 0.

[20] Rajeev Gopalakrishna, Eugene H Spafford, and Jan Vitek. 2005. Efficient intrusion detection using automaton inlining. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 18–31.

[21] Guofei Gu, Phillip A Porras, Vinod Yegneswaran, Martin W Fong, and Wenke Lee. 2007. BotHunter: Detecting Malware Infection Through IDS-Driven Dialog Correlation.. In *Proceedings of the USENIX Security Symposium*, Vol. 7. USENIX Association, Berkeley, CA, USA, 1–16.

[22] Zhongshu Gu, Kexin Pei, Qifan Wang, Luo Si, Xiangyu Zhang, and Dongyan Xu. 2014. LEAPS: Detecting Camouflaged Attacks with Statistical Learning Guided by Program Analysis. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE Computer Society, Washington, DC, USA, 491–502.

[23] Heartbleed 2014. The Heartbleed bug, http://heartbleed.com/. (2014).

[24] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. 2015. Automatic Generation of Data-oriented Exploits. In *Proceedings of the 24th USENIX Conference on Security Symposium*. USENIX Association, Berkeley, CA, USA, 177–192.

[25] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. 2016. Data-Oriented Programming: On the Expressiveness of Non-Control Data Attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 969–986.

[26] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. 2011. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*. ACM, ACM, New York, NY, USA, 43–58.

[27] N. Hubballi, S. Biswas, and S. Nandi. 2011. Sequencegram: n-gram modeling of system calls for program based anomaly detection. In *Proceedings of the International Conference on Communication Systems and Networks*. IEEE, Washington, DC, USA, 1–10.

[28] Hajime Inoue and Anil Somayaji. 2007. Lookahead pairs and full sequences: a tale of two anomaly detection methods. In *Proceedings of the Annual Symposium on Information Assurance*. ASIA, Albany, NY, USA, 9–19.

[29] Md Rafiqul Islam, Md Saiful Islam, and Morshed U Chowdhury. 2011. Detecting Unknown Anomalous Program Behavior Using API System Calls. In *Informatics Engineering and Information Science*. Springer, Hamburg, Germany, 383–394.

[30] Jafar Haadi Jafarian, Ali Abbasi, and Siavash Safaei Sheikhabadi. 2011. A gray-box DPDA-based intrusion detection technique using system-call monitoring. In *Proceedings of the Annual Collaboration, Electronic messaging, Anti-Abuse and Spam Conference*. ACM, New York, NY, USA, 1–12.

[31] Rohit Jalan and Arun Kejariwal. 2012. Trin-Trin: Who's Calling? A Pin-Based Dynamic Call Graph Extraction Framework. *International Journal of Parallel Programming* 40, 4 (2012), 410–442.

[32] Sandeep Karanth, Srivatsan Laxman, Prasad Naldurg, Ramarathnam Venkatesan, J. Lambert, and Jinwook Shin. 2010. *Pattern Mining for Future Attacks.* Technical Report MSR-TR-2010-100. Microsoft Research.

[33] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-production Failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15).* ACM, New York, NY, USA, 344–360.

[34] Andrew P Kosoresow and Steven A Hofmeyr. 1997. Intrusion detection via system call traces. *IEEE software* 14, 5 (1997), 35–42.

[35] Jeffrey P. Lanza. 2001. SSH CRC32 attack detection code contains remote integer overflow. (2001). Vulnerability Notes Database.

[36] Wenke Lee and Salvatore J Stolfo. 1998. Data mining approaches for intrusion detection. In *Proceedings of the USENIX Security Symposium.* USENIX Association, Berkeley, CA, USA, 6–6.

[37] Zhen Liu, Susan M Bridges, and Rayford B Vaughn. 2005. Combining static analysis and dynamic learning to build accurate intrusion detection models. In *Proceedings of IEEE International Workshop on Information Assurance*. IEEE Computer Society, Washington, DC, USA, 164–177.

[38] Federico Maggi, Matteo Matteucci, and Stefano Zanero. 2010. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing* 7, 4 (2010), 381–395.

[39] J. Sukarno Mertoguno. 2014. Human Decision Making Model for Autonomic Cyber Systems. *International Journal on Artificial Intelligence Tools* 23, 06 (2014), 1460023.

[40] David Moore, Colleen Shannon, Douglas J Brown, Geoffrey M Voelker, and Stefan Savage. 2006. Inferring internet Denial-of-Service activity. *ACM Transactions on Computer Systems* 24, 2 (2006), 115–139.

[41] James Newsome, Brad Karp, and Dawn Song. 2006. Paragraph: Thwarting Signature Learning by Training Maliciously. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID).* Springer Berlin Heidelberg, Berlin, Heidelberg, 81–105.

[42] Paradyn 2016. The Paradyn Project, http://www.paradyn.org/. (2016).

[43] K Peason. 1901. On lines and planes of closest fit to systems of point in space. *Philos. Mag.* 2 (1901), 559–572.

[44] Roberto Perdisci, Davide Ariu, Prahlad Fogla, Giorgio Giacinto, and Wenke Lee. 2009. McPAD: A multiple classifier system for accurate payload-based anomaly detection. *Computer Networks* 53, 6 (2009), 864–881.

[45] R. Perdisci, Guofei Gu, and Wenke Lee. 2006. Using an Ensemble of One-Class SVM Classifiers to Harden Payload-based Anomaly Detection Systems. In *Proceedings of the International Conference on Data Mining*. IEEE Computer Society, Washington, DC, USA, 488–498.

[46] Roberto Perdisci, Wenke Lee, and Nick Feamster. 2010. Behavioral Clustering of HTTP-based Malware and Signature Generation Using Malicious Network Traces. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation.* USENIX Association, Berkeley, CA, USA, 26–26.

[47] Geoffrey K. Pullum. 1983. Context-freeness and the Computer Processing of Human Languages. In *Proceedings of the annual meeting on Association for Computational Linguistics.* ACL, Stroudsburg, PA, USA, 1–6.

[48] Bernhard Schölkopf, Robert C Williamson, Alex J Smola, John Shawe-Taylor, and John C Platt. 1999. Support Vector Method for Novelty Detection. In *Proceedings of the annual conference on Neural Information Processing Systems*, Vol. 12. The MIT Press, Cambridge, MA, USA, 582–588.

[49] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. 2001. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy.* IEEE Computer Society, Washington, DC, USA, 144–155.

[50] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. 2004. On the Effectiveness of Address-space Randomization. In *Proceedings of the ACM Conference on Computer and Communications Security.* ACM, New York, NY, USA, 298–307.

[51] Monirul Sharif, Kapil Singh, Jonathon Giffin, and Wenke Lee. 2007. Understanding precision in host based intrusion detection. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*. Springer, Hamburg, Germany, 21–41.

[52] Xiaokui Shu and Danfeng Yao. 2016. Program Anomaly Detection: Methodology and Practices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16).* ACM, New York, NY, USA, 1853–1854.

[53] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. 2015. Unearthing Stealthy Program Attacks Buried in Extremely Long Execution Paths. In *Proceedings of the 2015 ACM Conference on Computer and Communications Security (CCS).* ACM, New York, NY, USA, 401–413.

[54] Xiaokui Shu, Danfeng Yao, and Barbara G. Ryder. 2015. A Formal Framework for Program Anomaly Detection. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID).* Springer, Hamburg, Germany, 270–292.

[55] Alexander Sotirov. 2007. Heap Feng Shui in JavaScript. (2007). Black Hat Europe.

[56] S.C. Sundaramurthy, J. McHugh, X.S. Ou, S.R. Rajagopalan, and M. Wesch. 2014. An Anthropological Approach to Studying CSIRTs. *IEEE Security & Privacy* 12, 5 (September 2014), 52–60.

[57] Systemtap 2006. SystemTap Overhead Test, https://sourceware.org/ml/systemtap/2006-q3/msg00146.html. (2006).

[58] Fredrik Valeur, Giovanni Vigna, Christopher Kruegel, and Richard A. Kemmerer. 2004. A Comprehensive Approach to Intrusion Detection Alert Correlation. *IEEE Transactions on Dependable and Secure Computing* 1, 3 (July 2004), 146–169.

[59] David Wagner and R Dean. 2001. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 156–168.

[60] David Wagner and Paolo Soto. 2002. Mimicry Attacks on Host-based Intrusion Detection Systems. In *Proceedings of the ACM Conference on Computer and Communications Security*. ACM, New York, NY, USA, 255–264.

[61] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. 1999. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, Washington, DC, USA, 133–145.

[62] K. Xu, K. Tian, D. Yao, and B. G. Ryder. 2016. A Sharper Sense of Self: Probabilistic Reasoning of Program Behaviors for Anomaly Detection with Context Sensitivity. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, Washington, DC, USA, 467–478.

[63] K. Xu, D. D. Yao, B. G. Ryder, and K. Tian. 2015. Probabilistic Program Modeling for High-Precision Anomaly Classification. In *Proceedings of the IEEE 28th Computer Security Foundations Symposium*. IEEE, Washington, DC, USA, 497–511.

[64] Nong Ye and X Li. 2000. A markov chain model of temporal behavior for anomaly detection. In *Proceedings of the 2000 IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop*, Vol. 166. IEEE, Washington, DC, USA, 169.

[65] Stefano Zanero. 2004. Behavioral intrusion detection. In *Computer and Information Sciences*. Springer, Hamburg, Germany, 657–666.

[66] Hao Zhang, Danfeng Yao, and Naren Ramakrishnan. 2014. Detection of stealthy malware activities with traffic causality and scalable triggering relation discovery. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. ACM, ACM, New York, NY, USA, 39–50.

[67] Hao Zhang, Danfeng Yao, Naren Ramakrishnan, and Zhibin Zhang. 2016. Causality reasoning about network events for detecting stealthy malware activities. *Computers & Security* 58 (2016), 180–198.

Received July 2016; revised February 2017; accepted May 2017