Unearthing Stealthy Program Attacks Buried in Extremely Long Execution Paths

Xiaokui Shu
Department of Computer Science, Virginia Tech
Blacksburg, VA 24060
subx@cs.vt.edu

Danfeng (Daphne) Yao
Department of Computer Science, Virginia Tech
Blacksburg, VA 24060
danfeng@cs.vt.edu

Naren Ramakrishnan
Department of Computer Science, Virginia Tech
Blacksburg, VA 24060
naren@cs.vt.edu

ABSTRACT

Modern stealthy exploits can achieve attack goals without introducing illegal control flows, e.g., tampering with non-control data and waiting for the modified data to propagate and alter the control flow legally. Existing program anomaly detection systems focusing on legal control flow attestation and short call sequence verification are inadequate to detect such stealthy attacks. In this paper, we point out the need to analyze program execution paths and discover event correlations in large-scale execution windows among millions of instructions. We propose an anomaly detection approach with two-stage machine learning algorithms to recognize diverse normal call-correlation patterns and detect program attacks at both inter- and intra-cluster levels. We implement a prototype of our approach and demonstrate its effectiveness against three real-world attacks and four synthetic anomalies with less than 0.01% false positive rates and 0.1~1.3 ms analysis overhead per behavior instance (1k to 50k function or system calls).

Categories and Subject Descriptors

K.6 [**Management Of Computing And Information Systems**]: Security and Protection; D.4 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

Intrusion Detection; Program Attack; Long Execution Path; Function Call; Event Correlation; Machine Learning

1. INTRODUCTION

Injecting library/system calls and tampering with return addresses on the stack are popular early-age exploit techniques. Modern exploits, however, are developed with more subtle control flow manipulation tactics to hide them from existing detection tools. One example is the `sshd` flag variable overwritten attack (an example of non-control data attacks [5]). An attacker overwrites a flag variable, which indicates the authentication result, before the authentication procedure. As a result, the attacker bypasses critical security control and logs in after a failed authentication.

Besides the aforementioned attack, stealthy attacks can also be constructed based on existing exploits. Wagner and Soto first diluted a compact exploit (several system calls) into a long sequence (hundreds of system calls) [46]. Kruegel et al. further advanced this approach by building an attack into an extremely long execution path [27]. In their proposed exploit, the attacker accomplishes one element of an attack vector, relinquishes the control of the target program, and waits for another opportunity (exploited vulnerability) to construct the next attack element. Therefore, the elements of the attack vector are buried in an extreme long execution path (millions of instructions). We refer stealthy attacks whose construction and/or consequence are buried into long execution paths and cannot be revealed by any small fragment of the entire path as *aberrant path attacks*.

Call-based program anomaly detection systems have been proposed as a general solution to detect program attacks without specifying attack signatures. Most existing program anomaly detection systems can be categorized into two detection paradigms: *short call sequence validation* and *first-order automaton transition verification*. The former is primarily based on deterministic [10, 11, 21] or probabilistic [14, 29] n-grams (short fragments of a long trace) verification. The latter verifies individual state transitions in legal control flows (a state refers to a system call plus the program counter [39], a system call plus the call stack [9,23], a user-space routine [16], or a code block [1]). Advanced approaches in these two paradigms adopt argument/data-flow analysis [3, 15, 16, 31], probabilistic measurement [18], and event frequency analysis [13, 14, 47].

Existing anomaly detection solutions are effective as long as an attack can be discovered in a small detection window on attack traces, e.g., an invalid n-gram or an illegal control flow transition (the latter can be accompanied by data-flow analysis). The aforementioned diluting attack [46] may be detected if it involves illegal control flows. However, there does not exist effective solutions for detecting general aberrant path attacks, because these attacks cannot be revealed in a small detection window on traces.

Mining correlations among arbitrary events in a large-scale execution window is the key to the detection of aber-

CCS'15, October 12–16, 2015, Denver, Colorado, USA.
 ISBN 978-1-4503-3832-5/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810103.2813654.

rant path attacks that are buried in long execution paths. The scale of the window may vary from thousands to millions of instructions. However, straightforward generalization of existing approaches is inadequate for large-scale execution window analysis because of two challenges described below.

Training scalability challenge: existing automaton-based methods are first-order and only verify state transition individually. One needs a linear bounded automaton or a Turing machine to enforce the relation among arbitrary events. The generalization results in exponential time complexity for training. n-gram based methods (e.g., lookahead pair, practical hidden Markov model) have a similar exponential convergence complexities in terms of n; large n (e.g., 40) usually leads to false positives due to insufficient training.

Behavior diversity challenge: real-world programs usually realize various functionalities, which result in diverse program behaviors within large-scale execution windows. The distance between a normal program behavior and an anomalous one can be less than the distance between two normal ones. The diversity of normal behaviors makes traditional single-threshold probabilistic methods (e.g., hidden Markov model, one-class SVM) difficult to fine-tune for achieving both a low false positive rate and a high detection rate.

To defend against aberrant path attacks, we propose a detection approach that analyzes program behaviors in large-scale execution windows. Our approach maps program behavior instances extracted from large-scale execution windows into data points in a high-dimensional detection space. It then leverages specifically designed machine learning techniques to *i)* recognize diverse program behaviors, *ii)* discover event correlations, and *iii)* detect anomalous program behaviors in various subspaces of the detection space.

In addition to the *binary representation of event relations* in an execution window, our approach further models *quantitative frequency relations among occurred events*. Some aberrant path attacks deliberately or unintentionally result in anomalous event frequency relations, e.g., Denial of Service (DoS), directory harvest attack. The advantage of modeling frequency relations over individual event frequencies (used in existing anomaly detection [13]) is the low false positive rates in case of program/service workload variation.

The contributions of our work are summarized as follows.

- We study the characteristics of aberrant path attacks and identify the need to analyze program behaviors in large-scale execution windows. We present a security model for efficient program behavior analysis through event correlations in large-scale execution windows. The security model covers the detection of two types of anomalous program behaviors abstracted from four known categories of aberrant path attacks. The first type contains events (and their corresponding control-flow segments) that are incompatible in a single large-scale execution window, e.g., non-control data attacks. The second type contains aberrant relations among event occurrence frequencies, e.g., service abuse attacks.

- We design a two-stage detection approach to discover anomalous event correlations in large-scale execution windows and detect aberrant path attacks. Our approach contains a *constrained* agglomerative clustering algorithm for addressing the behavior diversity challenge and dividing the detection problem into subproblems. Our approach addresses the scalability challenge by utilizing fixed-size profiling matrices and by estimating normal behavior patterns from an incomplete training set through probabilistic methods in each cluster. The unique two-stage design of our approach enables effective detections of *i)* legal-but-incompatible control-flow segments and *ii)* aberrant event occurrence frequency relations at inter- and intra-cluster levels.

- We implement a prototype of our approach on Linux and evaluate its detection capability, accuracy, and performance with `sshd`, `libpcre` and `sendmail`. The evaluation contains over 22,000 normal profiles and over 800 attack traces. Our approach successfully detects all attack attempts with less than 0.01% false positive rates. We demonstrate the high detection accuracy of our clustering design through the detection of four types of synthetic anomalies. Our prototype takes 0.3~1.3 ms to analyze a single program behavior instance, which contains 1k to 50k function/system call events.

```
 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;
        ...
```

Figure 1: `sshd` flag variable overwritten attack [5].

2. SECURITY MODEL

We describe the attack model, explain our security goals, and discuss three basic solutions toward the goals.

2.1 Aberrant Path Attack

We aim to detect aberrant path attacks, which contain infeasible/inconsistent/aberrant execution paths but obey legitimate control-flow graphs. Aberrant path attacks can evade existing detection mechanisms because of the following properties of the attacks:

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

Aberrant path attacks are realistic threats and gain popularity since early-age attacks have been efficiently detected and blocked. Concrete aberrant path attack examples are:

a) *Non-control data attacks* hijack programs without manipulating their control data (data loaded into program counter in an execution, e.g., return addresses). One such attack, first described by Chen et al. [5], takes advantage of an integer overflow vulnerability found in several implementations of the `SSH1` protocol [28]. Illustrated in Fig. 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.

b) *Workflow violation attacks* can be used to bypass access control [6], leak critical information, disable a service (e.g., trigger a deadlock), etc. One example is *presentation layer access control bypass* in web applications. If the authentication is only enforced by the presentation layer, an attacker can directly access the business logic layer (below presentation layer) and read/write data.

c) *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 [41]), seeks addresses of useful code and data fragments (e.g., ASLR probing [40]), and/or triggers particular race conditions.

d) *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 [19]), and financial interest (e.g., click fraud) of target services.

2.2 Anomalous Program Behaviors within Large-scale Execution Windows

Aberrant path attacks cannot be detected by analyzing events in small windows on program traces. We define semantically meaningful execution windows and unearth aberrant path attacks in large-scale execution windows.

Definition 2.1. *An execution window W is the entire or an autonomous portion of a transactional or continuous program execution.*

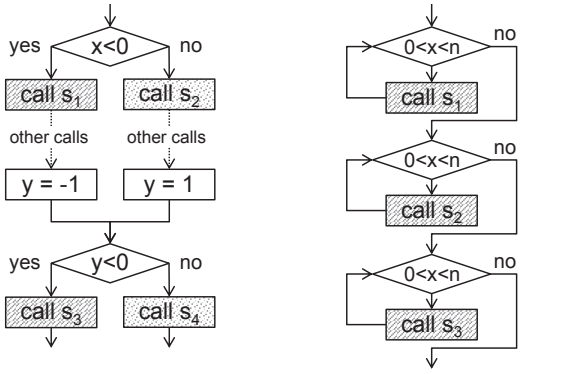
Execution windows can be partitioned based on boundaries of program functionalities, e.g., login, session handling, etc. Since aberrant path attacks can lead to delayed attack consequences, e.g., non-control data attacks, the analysis should be performed on large-scale execution windows. One such window could contain tens of thousands of system calls and hundreds of times more function calls.

We give some examples of practical large-scale execution window partitioning for security analysis purposes:

i) partitioning by routines/procedures/functions,
ii) partitioning by threads or forked processes,
iii) partitioning by activity intervals, e.g., `sleep()`,
iv) an entire execution of a small program.

In large-scale execution windows, we abstract two common anomalous behavior patterns of aberrant path attacks.

1. *Montage anomaly* is an anomalous program behavior composed of multiple legitimate control flow fragments that are incompatible in a single execution.



(a) The executions of s_1 and s_3 occur in the same run, similarly for s_2 and s_4.

(b) s_1, s_2 and s_3 occur at the same frequency in a run.

Figure 2: Examples of control flows that illustrate event co-occurrence patterns and occurrence frequency relations.

One example of a montage anomaly is the `sshd` flag variable overwritten attack presented in Fig. 1. The attack incurs an execution path that contains two incompatible execution segments: *i)* fail-auth handling (line 13-16) and *ii)* pass-auth execution (line 18-).

2. *Frequency anomaly* is an anomalous program behavior with aberrant ratios/relations between/among event occurrence frequencies. Normal relations among frequencies are established by: *i)* mathematical relations among induction variables that are specified in the binary (e.g., Fig. 2b), and *ii)* normal usage patterns of the program.

   One example of a frequency anomaly is a *directory harvest attack* against a mail server. The attack probes legitimate usernames on the server with a batch of emails targeting possible users. The attack results in an aberrant ratio between event frequencies in the server's handling procedures of existent/nonexistent receivers.

   Sometimes an event occurrence frequency alone can indicate an attack, e.g., DoS. However, the workload of a real-world service may vary rapidly, and the individual frequencies are imprecise to model program behaviors.

2.3 Security Goals

The key to the detection of montage anomalies and frequency anomalies is to model and analyze the *relations among control-flow segments that occur in a large-scale execution window*. We further deduce two practical security goals for detecting aberrant path attacks. The deduction is based on the fact that *events* (e.g., call, jmp, or generic instructions) in dynamic program traces mark/indicate the control-flow segment to which they belong.

1. *Event co-occurrence analysis* examines the patterns of co-occurred events in a large-scale execution window[1]. We illustrate an event co-occurrence analysis in Fig. 2a. Rules should be learned that events $\langle s_1, s_3 \rangle$ or $\langle s_2, s_4 \rangle$ always occur together, but not $\langle s_1, s_4 \rangle$ or $\langle s_2, s_3 \rangle$.

2. *Event occurrence frequency analysis* examines the event occurrence frequencies and the relations among them.

[1]We define the co-occurrence of events in the scope of an execution window, not essentially at the same time.

For instance, s_1, s_2 and s_3 always occur at the same frequency in Fig. 2b. Another type of event occurrence frequency relation is generated utterly due to specific usage patterns (mail server example in Sect. 2.2), which can be only learned from dynamic traces.

2.4 Basic Solutions and Their Inadequacy

There are several straightforward solutions providing event co-occurrence and occurrence frequency analysis. We point out their limitations, which help motivate our work.

Basic Solution I: One can utilize a large n in an n-gram approach (either deterministic approaches, e.g., [11], or probabilistic approaches, e.g., hidden Markov model [14, 47]). This approach detects aberrant path attacks because long n-grams are large execution windows. However, it results in exponential training convergence complexity and storage complexity. Unless the detection system is trained with huge number of normal traces, which is exponential to n, a large portion of normal traces will be detected as anomalous. The exponential convergence complexity explains why no n-gram approach employs $n > 40$ in practice [10].

Basic Solution II: One can patch existing solutions with frequency analysis components to detect some aberrant path attacks, e.g., DoS. The possibility has been explored by Hubballi et al. on n-grams [20] and Frossi et al. on automata state transitions [13]. Their solutions successfully detect DoS attacks through unusually high frequencies of particular n-grams and individual automata state transitions. However, the underlying detection paradigms restrict the solutions from correlating arbitrary events in a long trace. Thus, their solutions do not detect general aberrant path attacks.

Basic Solution III: One can perform episodes mining within large-scale execution windows. It extends existing *frequent episode mining* [25, 29] by extracting episodes (featured subsequences) at all frequencies so that infrequent-but-normal behaviors can be characterized. In order to analyze all episodes (the power set of events in a large-scale execution window), this approach faces a similar exponential complexity of training convergence as Basic Solution I.

3. OVERVIEW OF OUR APPROACH

We present an overview of our approach analyzing event co-occurrence and event occurrence frequencies in large-scale execution windows. We develop a constrained agglomerative clustering algorithm to overcome the behavior diversity challenge. We develop a compact and fixed-length matrix representation to overcome the scalability problem for storing variable-length trace segments. We utilize probabilistic methods to estimate normal behaviors in an incomplete training dataset for overcoming the training scalability issue.

3.1 Profiling Program Behaviors

We design our approach to expose user-space program activities (executed control flow segments) via `call` instructions. `call` and `ret`[2] are responsible for call stack changes and provide a natural boundary for determining execution windows as discussed in Section 2.2.

[2]`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 execution window partitioning.

We denote the overall activity of a program P within an execution window W as a behavior instance b. Instance b recorded in a program trace 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 W.*

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

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

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, our approach 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 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 of Our Approach

Our approach consists of two complementary stages of modeling and detection where montage/frequency anomalies are detected in the first/second stage, respectively.

The first stage models the binary representation of event co-occurrences in a large-scale execution window via event co-occurrence matrix O. It performs event co-occurrence analysis against montage anomalies. 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 large-scale execution window via transition frequency matrix F. It performs event occurrence frequency analysis against frequency anomalies. It consists of a training operation Intra-cluster Modeling and a detection operation Occurrence Frequency Analysis.

We illustrate the architecture of our approach in Fig. 3 and brief the functionalities of each operation below.

1. Behavior Profiling recognizes target execution windows $\{W_1, W_2, \dots\}$ in traces and profiles b from each W into O and F. Symbols in F and O are retrieved via static program analysis or system call table lookup.

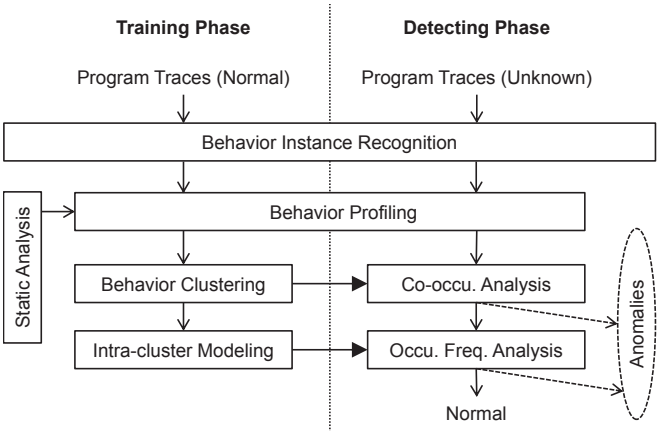


Figure 3: Information flows among operations in two stages and two phases of our program anomaly detection approach.

2. Behavior Clustering is a training operation. It takes in all normal behavior instances $\{b_1, b_2, \dots\}$ and outputs a set of behavior clusters $\mathbb{C} = \{C_1, C_2, \dots\}$ where $C_i = \{b_{i_1}, b_{i_2}, \dots\}$.

3. Intra-cluster Modeling is a training operation. It is performed in each cluster. It takes in all normal behavior instances $\{b_{i_1}, b_{i_2}, \dots\}$ 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 $\mathbb{C}$ to seek montage anomalies. If behavior instance b is normal, it reduces the detection problem to subproblems within a set of behavior clusters $\mathbb{C}_b = \{C_{b_1}, C_{b_2}, \dots\}$, in which b closely fits.

5. Occurrence Frequency Analysis is an intra-cluster detection operation that analyzes F (of b) in each C_b to seek frequency anomalies. 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 training/modeling and detection operations in our two-stage approach. The key to the first stage is a customized clustering algorithm, which differentiates diverse program behaviors and divides the detection problem into subproblems. Based on the clustering, inter-/intra-cluster detection is performed 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. 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.

Algorithm 1 Constrained agglomerative clustering for grouping similar program behavior instances.

Require: a set of normal program behavior instances B and a termination threshold T_d. $dist()$ is the distance function between behaviors/clusters. $pen()$ is the penalty function for long tail elimination.
Ensure: a set of behavior clusters $\mathbb{C}$.

```
1: h ← ∅_heap
2: v ← ∅_hashtable
3: V ← ∅_set
4: for all b ∈ B do
5:     O ← O_b
6:     v[O] ← v[O] + 1
7:     for all O′ ∈ V do
8:         d_p ← dist(O, O′) × pen(v[O], v[O′])
9:         push ⟨d_p, O, v[O], O′, v[O′]⟩ onto h
10:    end for
11:    add O to V
12: end for
13: while h ≠ ∅_heap do
14:    pop ⟨d_p, O_1, v_O1, O_2, v_O2⟩ from h
15:    break if d_p > T_d
16:    if O_1 ∈ V and O_2 ∈ V then
17:        continue if v_O1 < v[O_1] or v_O2 < v[O_2]
18:        O ← O_1 OR O_2
19:        v[O] ← v[O_1] + v[O_2]
20:        remove O_1 from V
21:        remove O_2 from V
22:        for all O′ ∈ V do
23:            d_p ← dist(O, O′) × pen(v[O], v[O′])
24:            push ⟨d_p, O, v[O], O′, v[O′]⟩ onto h
25:        end for
26:        add O to V
27:    end if
28: end while
29: w[O] ← ∅_set for all O ∈ V
30: for all b ∈ B do
31:    O ← O_b
32:    m ← MAXINT
33:    for all O′ ∈ V do
34:        if O OR O′ = O′ then
35:            if dist(O, O′) < m then
36:                m ← dist(O, O′)
37:                V′ ← {O′}
38:            else if dist(O, O′) = m then
39:                add O′ to V′
40:            end if
41:        end if
42:    end for
43:    add b to w[O] for all O ∈ V′
44: end for
45: ℂ ← {w[O] for all O ∈ V}
```

Our algorithm (Algorithm 1) clusters program behavior instances based on the co-occurred events shared among instances. To deal with the borderline behavior issue, we alter the standard process into a two-step process: *i)* generate scopes of clusters in an agglomerative way (line 13-28), and *ii)* add behavior instances to generated clusters (line 30-44).

We use a lazily updated heap h in Algorithm 1 to minimize the calculation and sorting of distances between intermediate clusters. We perform 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 $C = \{b_i \mid 0 \le i \le k\}$ is represented by its event co-occurrence matrix O_C. O_C records occurred events in any behavior instances in C. It is calculated using

(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.

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

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.

We explain the unique features of our constrained agglomerative clustering algorithm over the standard design:

- *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 9 and line 24). An entry is abandoned if any of its two clusters are found deprecated at the moment the entry is popped from h (line 17).

- *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 this type of borderline behaviors correctly, our clustering algorithm duplicates b in every cluster, which b may belong to (line 30-44). 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 montage 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 $O_b \texttt{ OR } O_C = O_C$ where O_C and O_b are the event co-occurrence matrices of C and b. The detection process searches for all clusters in which b fits. If this set of clusters is not empty, distances between b and each cluster in this set are calculated using (3). The clusters with the nearest distance (there could be more than one cluster) are selected as $\mathbb{C}_b$.

4.3 Intra-cluster Modeling (Training)

Within a cluster C, our approach 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 [38], 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)[3] or *principle component analysis* (PCA)[4] 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)[5], 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 frequency anomalies: *i)* deviant relations among multiple event occurrence frequencies, and/or *ii)* aberrant occurrence

[3]FVA selects dimensions/events with larger-than-threshold frequency variances across all behavior instances in C.
[4]PCA selects linear combinations of dimensions/events with larger-than-threshold frequency variances, which is a generalization of FVA.
[5]Multiple functions have been tested for selection.

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 (5).

$$\exists C \in \mathbb{C}\ N_{clt}(b, C) \Rightarrow b \text{ is normal} \tag{5}$$

The detection inside C is performed with 3 rules, and the result is aggregated into $N_{clt}(b, C)$.

$$N_{clt}(b, C) = \begin{cases} \texttt{True} & \text{normal by all 3 rules} \\ \texttt{False} & \text{anomalous by any rule} \end{cases} \tag{6}$$

- *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 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 range model detection.* Events in b with zero or near zero variances are tested against the range model (the deterministic method) 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 generates false negatives. It sacrifices the detection rate for reducing false alarms in insufficiently trained clusters.

4.5 Discussion

Our program anomaly detection approach is a context-sensitive language parser from the formal language perspective, i.e., Bach language parser [37]. In comparison, existing automata methods are at most context-free language parsers (pushdown automata methods) [8]. n-gram methods are regular language parsers (finite state machine equivalents [46]). Existing probabilistic methods are stochastic languages parsers (probabilistic regular language parsers).

A context-sensitive language parser is more precise than a context-free language parser or a regular language parser in theory. It is accepted by a linear bounded automaton (LBA), which is a restricted Turing machine with a finite tape. The advantage of a context-sensitive parser is its ability to characterize cross-serial dependencies, or to correlate far away events in a long program trace.

Our approach explores the possibility to construct an efficient program anomaly detection approach on the context-sensitive language level. Potential mimicry attacks could be constructed to exploit the gap between Bach and the most precise program execution description. However, it is more difficult to do so than constructing mimicry attacks against regular or context-free language level detection tools. For example, padding is a simple means to construct regular language level mimicry attacks, and our approach can detect padding attacks. Our analysis characterizes whether two function calls should occur in one execution window, so padding rarely occurred calls can be detected. Our approach recognizes the ratios between call pairs in one execution window. Thus, excessive padding elements can be discovered.

Potential mimicry attacks may exploit the monitoring granularity of a detection approach. Our current approach utilizes `call` instructions to mark control-flow segments, which can be generalized to any instruction for detecting mimicry attacks that do not involve `call` in any part of their long attack paths.

5. IMPLEMENTATION

We implement a prototype of our detection approach on Linux (Fedora 21, kernel 3.19.3). The static analysis is realized through `C` (`ParseAPI` [34]). 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 (Sect. 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 execution windows 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 large-scale execution windows 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 execution window partitioning via activity intervals when the program is monitored as a black box.

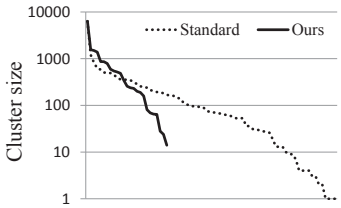
6. EVALUATIONS

To verify the detection capability of our approach, we test our prototype against different types of aberrant path attacks (Sect. 6.2). We investigate its detection accuracy using real and synthetic program traces (Sect. 6.3). We evaluate the performance of our prototype with different tracing and detection options (Sect. 6.4).

Table 1: The profile information of programs/libraries and statistics of normal profiles.

		Profile Overview			Average Single Normal Profile		
Program	**Version**	**Events in Profile**	**Execution Window**	**#(N.P.)**	**#(Symbols)**	**#(Event)**	**#(U.E.)**
`sshd`	1.2.30	function calls	routine boundary	4800	415	34511	180
`libpcre`	8.32	function calls	library call	11027	79	44893	45
`sendmail`	8.14.7	system calls†	continuous operation	6579	350	1134	213

N.P. is short for *normal profile*. U.E. is short for *unique event*.
†Function calls are not traced due to its complex process spawning logic. Customization of our Pintool is needed to trace them.



(a) `libpcre` cluster size distribution (sorted in descending order).

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

(b) Overview of program behavior clustering.

Figure 4: Clustering of program behavior instances.

6.1 Experiment Setup

We study three programs/libraries (Table 1) in distinct categories. We demonstrate that our approach is a versatile detection solution to be applied to programs and dynamic libraries with various large-scale execution window definitions and event definitions. We detail the programs/libraries and their training dataset (normal profiles) below.

`[sshd]` Execution window definition: program activities of `sshd` within routine `do_authentication()`. The routine `do_authentication()` is called in a forked thread after a client initializes its connection to `sshd`. All session activities are within the execution window if the authentication is passed. 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]` Execution window definition: program activities of `libpcre` when a library call is made. Library calls are triggered through `grep -P`. Over 10,000 normal tests are used from the `libpcre` package.

`[sendmail]` Execution window 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 our detection approach 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 Fig. 4b[6]. $|\mathbb{C}|$ denotes the number of clusters computed with the specific T_d. In Fig. 4a, we demonstrate the effectiveness of our constrained agglomerative clustering algorithm to eliminate tiny clusters. The standard agglomerative clustering approach results in a long-tail distribution of cluster sizes shown in Fig. 4a.

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 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[7] 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 Discovering Real-World Attacks

We reproduce three known aberrant path attacks to test the detection capability of our approach. Our detection approach detects all attack attempts with less than 0.0001 false positive rate. The overview of the attacks and detection results are presented in Table 2.

6.2.1 Flag Variable Overwritten Attack

Flag variable overwritten attack 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. [5]. We describe the attack in Sect. 2.1, bullet (a) and in Fig. 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. For each attack attempt, 128 random commands are executed after a successful login.

Our approach (configured at FPR^u 0.0001) successfully detects all attack attempts in inter-cluster detection (Co-occurrence Analysis)[8]. We present normal and attack traces inside the execution window (selected routine `do_authentication()`) in Fig. 5 to illustrate the detection.

[6]The value is empirically chosen to keep a balance between an effective recognition of diverse behaviors and an adequate elimination of tiny clusters.

[7]The distance can be positive or negative. More details are specified in Rule 1 (Sect. 4.4).

[8]One-class SVM in Occurrence Frequency Analysis only detects 3.8% attack attempts if used alone.

Table 2: Overview of reproduced attacks and detection results.

Attack Name	Target	Attack Settings	#(A.)	D.R.	FPR^u
flag variable overwritten attack	`sshd`	an inline virtual exploit that matches a username	800	100%	0.0001
Regular expression Denial of Service	`libpcre`	3 deleterious patterns paired with 8-23 input strings	46	100%	0.0001
directory harvest attack	`sendmail`	probing batch sizes: 8, 16, 32, 64, 100, 200, and 400	14	100%	0.0001

A. is short for *attack attempt*. D.R. is short for *detection rate*. FPR^u is the false positive rate upper bound (details in Sect. 6.1).

Normal[a]	**Normal**[b]	**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
...	...	...

[a] A successfully authenticated session.
[b] A failed (wrong password) authentication.
"caller > callee" denotes a function call.
Routine names are abbreviated to save space.

Figure 5: Samples of normal and anomalous `sshd` traces.

Table 3: Deleterious patterns used in ReDoS attacks.

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

In Fig. 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 as a montage anomaly: 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 a single execution window.

In the traces, there are identical 218 `call` events including library routines (36 `call`s excluding library ones) between the third line and the last line in Fig. 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 that n-gram models with a large n is difficult to converge at training. We do not test automaton-based detection because they cannot detect the attack in theory. The attack does not contain any illegal function calls.

6.2.2 Regular Expression Denial of Service

Regular expression Denial of Service (ReDoS) 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, but the regex engine hangs due to the extreme complexity.

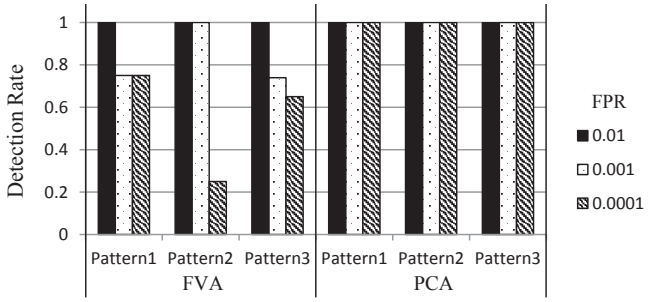


Figure 6: Detection rates of ReDoS attacks.

We produce 46 ReDoS attack attempts targeting `libpcre`[9]. Three deleterious patterns are used (Table 3). 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 approach with both PCA and FVA feature selection (Sect. 4.3, the probabilistic method, bullet b). The detection results (Fig. 6) show that our approach configured with PCA is more sensitive than it configured with FVA. Our approach (with PCA) detects all attack attempts at different FPRs[10]. 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.

6.2.3 Directory Harvest Attack

Directory harvest attack (DHA) 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 2). 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[10]. DHA attacks are detected in intra-cluster detection (Occurrence Frequency Analysis) by the probabilistic method, i.e., ν-SVM. The attacks bypass the inter-

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

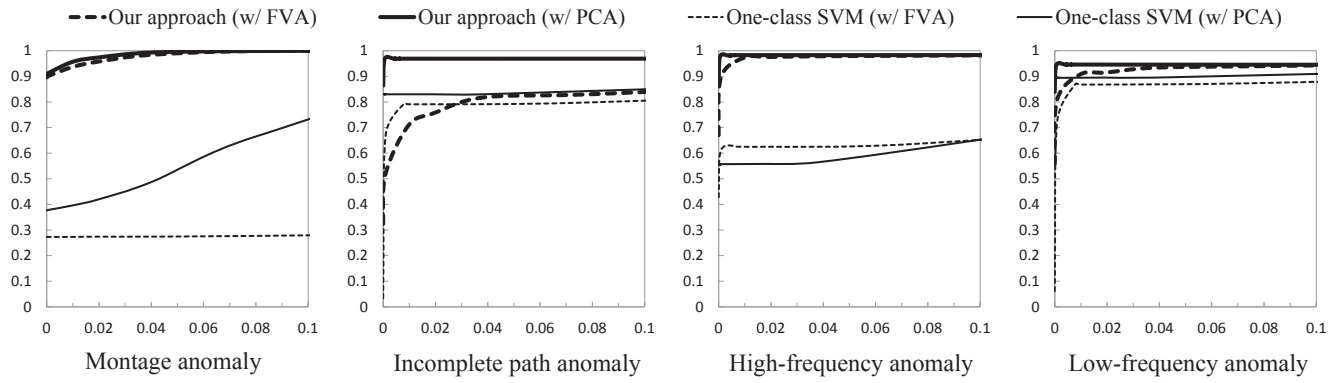


Figure 7: `libpcre` ROC of our approach and basic one-class SVM. X-axis is false positive rate, and y-axis is detection rate.

cluster detection (Co-occurrence Analysis) because invalid usernames occur in normal training dataset.

This experiment demonstrates that our approach 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 an execution window 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 function calls.

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 (Sect. 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.

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

We present the detection accuracy results on `libpcre` in Fig. 7, which has the most complicated behavior patterns among the three studied programs/libraries[11]. In any subfigure of Fig. 7, 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 denote an *anomaly* result as a *positive*.

Fig. 7 shows the effectiveness of our clustering design. The detection rate of our prototype (with PCA[12]) 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.

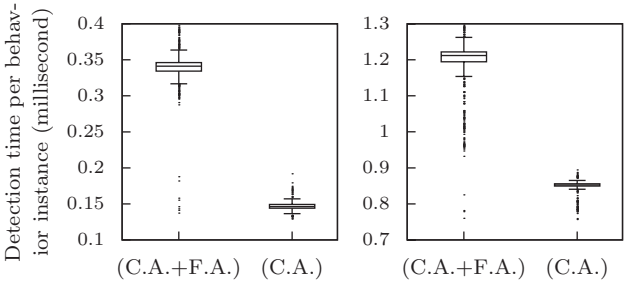
6.4 Performance Analysis

Although performance is not a critical issue for the training phase, a fast and efficient detection is important for enabling real-time protection and minimizing negative user experience [32]. The overall overhead of a program anomaly detection system comes from *tracing* and *analysis* in general.

We evaluate the performance of our 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`: 11027, `sendmail`: 6579) to collect overhead for *inter-cluster detection alone* and *the combination of inter- and intra-cluster detection*[13]. The analysis of each behavior instance is repeated 1,000 times to obtain a fair timing. The performance results in Fig. 8 illustrate that

- 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 1).
- The analysis overhead is positively correlated with the number of unique events in a profile (Table 1), which is due to our DOK implementation of profile matrices.
- Montage anomalies takes less time to detect than frequency anomalies, because they are detected at the first stage (Co-occurrence Analysis).

[11]Results of the other two programs share similar characteristics as `libpcre` and are not presented.
[12]PCA proves itself more accurate than FVA in Fig. 7.
[13]PCA is used for feature selection. FVA (results omitted) yields a lower overhead due to its simplicity.



(a) libpcre (b) sendmail

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

Figure 8: Detection (analysis) overhead of our approach.

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` [24]. Advanced tracing techniques, e.g., probe mode pintool, branch target store [48], etc., can potentially reduce the tracing overhead to less than 10% toward a real-time detection system.

Another choice to deploy our detection solution is to profile program behaviors through system calls as we demonstrate using `sendmail`. System calls can be traced through `SystemTap` with near-zero overhead [43], but it sacrifices the capability to reveal user-space program activities and downgrades the modeling/detection accuracy.

Our approach can support offline detection or forensics of program attacks, in which case accuracy is the main concern instead of performance [42]. Our Pintool enables analysts to locate anomalies within execution windows, and our matrices provide caller information for individual function calls. This information helps analysts quickly reduce false alarms and locate vulnerable code segments.

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. RELATED WORK

Conventional program anomaly detection systems (aka host-based intrusion detection systems) follow Denning's intrusion detection vision [7]. 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. [11]; and *ii)* automaton transition verification, which was first described by Kosoresow and Hofmeyr [26] (DFA) and formalized by Sekar et al. [39] (FSA) and Wagner and Dean [45] (NDPDA).

The basic n-gram model was further studied in [10,21] and several advanced forms of it were developed, e.g., machine learning models [22, 29] and hidden Markov models [14, 47].

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. 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 [46].

Although Wagner and Soto's attack evades the detection from n-gram methods, it is system-call-level tactics, and it may introduce *illegal control flows*, which can be captured by pushdown automaton (PDA) methods [8, 9, 15, 16]. This mimicry attack could also involve *anomalous call arguments*, which can be detected by argument analysis [15, 31].

Research on automaton detection started with the goal of performing trace analysis on a large scale. However, all existing automaton models are equivalents to FSA/PDA. They only verify state transitions individually, i.e., they are first-order models. Program counter and call stack information were used to help precisely define each state (a system call) in an automaton [8, 9, 39]. Function calls/returns are included as automaton states in the Dyck model [16]. Models combining static and dynamic analysis were developed [23,30], and individual transition frequencies have been employed to detect DoS attacks [13].

All existing automaton detection methods cannot be directly used for detecting aberrant path anomalies, as explained earlier in the paper. Existing detection methods lack the ability to correlate events in different control-flow segments in a large-scale execution window. An automaton that is capable to do so would have an exponential complexity in term of training overhead and storage.

The relation among events that occur far away has not been systematically studied in the literature. In this paper, we formalize the problem of event correlation analysis within large-scale execution windows and bring forward the first solution that correlates events in a large-scale execution window for anomaly detection purpose.

Clustering and classification techniques have been widely used in malware classification [2, 12, 25, 36]. 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 [44]. Perdisci et al. proposed 2_v-gram scheme to discover related bytes v positions apart in traffic payload [35]. Gu et al. developed a system to correlate temporal network events for detecting botnets under specific bot behavior hypotheses [17]. 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 [33], 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.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we studied aberrant path attacks and designed a two-stage anomaly detection approach to unearthing these attacks from extreme long program traces. The significance of our work is the new capability to efficiently discover subtle program inconsistencies and anomalies that occur far apart. Our work advances the state-of-the-art program anomaly detection by demonstrating the effectiveness of large-scale program behavioral modeling and enforcement against runtime anomalies that are buried in extremely long execution paths. In future work, we plan to adopt advanced dynamic tracing techniques and build real-time security incidence response systems on top of our detection solution.

9. ACKNOWLEDGMENTS

This work has been supported by ONR grant N00014-13-1-0016 and ARO YIP W911NF-14-1-0535. The authors would like to thank Barbara Ryder for her feedback on the work. The authors would like to thank Changhee Jung and Dongyoon Lee for their comments on performance analysis.

10. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[2] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Krügel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the Network and Distributed System Security Symposium*, 2009.

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

[4] D. Brumley, D. X. Song, T. cker Chiueh, R. Johnson, and H. Lin. RICH: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2007.

[5] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the USENIX Security Symposium*, volume 14, pages 12–12, 2005.

[6] M. Cova, D. Balzarotti, V. Felmetsger, and G. Vigna. 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*, pages 63–86, 2007.

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

[8] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 194–208, 2004.

[9] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 62–75, 2003.

[10] S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *Proceedings of the Annual Computer Security Applications Conference*, pages 418–430, 2008.

[11] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

[12] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 45–60, 2010.

[13] A. Frossi, F. Maggi, G. L. Rizzo, and S. Zanero. Selecting and improving system call models for anomaly detection. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 206–223. Springer, 2009.

[14] D. Gao, M. K. Reiter, and D. Song. Behavioral distance measurement using hidden Markov models. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*, pages 19–40, 2006.

[15] J. T. Giffin, D. Dagon, S. Jha, W. Lee, and B. P. Miller. Environment-sensitive intrusion detection. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses*, pages 185–206, 2006.

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

[17] G. Gu, P. A. Porras, V. Yegneswaran, M. W. Fong, and W. Lee. BotHunter: Detecting malware infection through IDS-driven dialog correlation. In *Proceedings of the USENIX Security Symposium*, volume 7, pages 1–16, 2007.

[18] Z. Gu, K. Pei, Q. Wang, L. Si, X. Zhang, and D. Xu. LEAPS: Detecting camouflaged attacks with statistical learning guided by program analysis. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 491–502, 2014.

[19] The heartbleed bug, `http://heartbleed.com/`.

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

[21] H. Inoue and A. Somayaji. Lookahead pairs and full sequences: a tale of two anomaly detection methods. In *Proceedings of the Annual Symposium on Information Assurance*, pages 9–19, 2007.

[22] M. R. Islam, M. S. Islam, and M. U. Chowdhury. Detecting unknown anomalous program behavior using API system calls. In *Informatics Engineering and Information Science*, pages 383–394. Springer Berlin Heidelberg, 2011.

[23] J. H. Jafarian, A. Abbasi, and S. S. Sheikhabadi. 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*, pages 1–12, 2011.

[24] R. Jalan and A. Kejariwal. Trin-Trin: Who's calling? a Pin-based dynamic call graph extraction framework. *International Journal of Parallel Programming*, 40(4):410–442, 2012.

[25] S. Karanth, S. Laxman, P. Naldurg, R. Venkatesan, J. Lambert, and J. Shin. Pattern mining for future attacks. Technical Report MSR-TR-2010-100, Microsoft Research, 2010.

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

[27] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the USENIX Security Symposium*, pages 11–11, 2005.

[28] J. P. Lanza. SSH CRC32 attack detection code contains remote integer overflow. *Vulnerability Notes Database*, 2001.

[29] W. Lee and S. J. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the USENIX Security Symposium*, pages 6–6, 1998.

[30] Z. Liu, S. M. Bridges, and R. B. Vaughn. Combining static analysis and dynamic learning to build accurate intrusion detection models. In *Proceedings of IEEE International Workshop on Information Assurance*, pages 164–177, 2005.

[31] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395, 2010.

[32] J. S. Mertoguno. Human decision making model for autonomic cyber systems. *International Journal on Artificial Intelligence Tools*, 23(06):1460023, 2014.

[33] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage. Inferring internet Denial-of-Service activity. *ACM Transactions on Computer Systems*, 24(2):115–139, 2006.

[34] The paradyn project, `http://www.paradyn.org/`.

[35] R. Perdisci, G. Gu, and W. Lee. Using an ensemble of one-class SVM classifiers to harden payload-based anomaly detection systems. In *Proceedings of the International Conference on Data Mining*, pages 488–498, December 2006.

[36] R. Perdisci, W. Lee, and N. Feamster. 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*, pages 26–26, 2010.

[37] G. K. Pullum. Context-freeness and the computer processing of human languages. In *Proceedings of the annual meeting on Association for Computational Linguistics*, pages 1–6, Stroudsburg, PA, USA, 1983.

[38] B. Schölkopf, R. C. Williamson, A. J. Smola, J. Shawe-Taylor, and J. C. Platt. Support vector method for novelty detection. In *Proceedings of the annual conference on Neural Information Processing Systems*, volume 12, pages 582–588, 1999.

[39] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 144–155, 2001.

[40] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 298–307, 2004.

[41] A. Sotirov. Heap Feng Shui in JavaScript. *Black Hat Europe*, 2007.

[42] S. Sundaramurthy, J. McHugh, X. Ou, S. Rajagopalan, and M. Wesch. An anthropological approach to studying CSIRTs. *IEEE Security & Privacy*, 12(5):52–60, September 2014.

[43] Systemtap overhead test, `https://sourceware.org/ml/systemtap/2006-q3/msg00146.html`.

[44] F. Valeur, G. Vigna, C. Kruegel, and R. A. Kemmerer. A comprehensive approach to intrusion detection alert correlation. *IEEE Transactions on Dependable and Secure Computing*, 1(3):146–169, July 2004.

[45] D. Wagner and R. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–168, 2001.

[46] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 255–264, 2002.

[47] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

[48] Y. Xia, Y. Liu, H. Chen, and B. Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *Proceedings of the Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12, June 2012.