Automated caching of behavioral patterns for efficient run-time monitoring

Natalia Stakhanova[1] Samik Basu[1] Robyn R. Lutz[1,2*] Johnny Wong[1]

[1]*Department of Computer Science*
Iowa State University
Ames, IA 50011 USA

[2]*Jet Propulsion Laboratory*
California Institute of Technology
Pasadena, CA 91109 USA

{*ndubrov, sbasu, rlutz, wong*}*@iastate.edu*

Abstract

Run-time monitoring is a powerful approach for dynamically detecting faults or malicious activity of software systems. However, there are often two obstacles to the implementation of this approach in practice: (1) that developing correct and/or faulty behavioral patterns can be a difficult, labor-intensive process, and (2) that use of such pattern-monitoring must provide rapid turn-around or response time. We present a novel data structure, called extended action graph, and associated algorithms to overcome these drawbacks. At its core, our technique relies on effectively identifying and caching specifications from (correct/faulty) patterns learned via machine-learning algorithm. We describe the design and implementation of our technique and show its practical applicability in the domain of security monitoring of sendmail software.

1. Introduction

Run-time monitoring is a proven technique for enhancing the dependability of a system. It observes the behavior of the system during execution and detects anomalous deviations from normal or expected behavior. Early indications of these deviations from expected behavior are frequently useful from a dependability perspective. They may indicate possible movement of the system from a safe to an unsafe state (e.g., from an aerodynamically stable to unstable state [16]), from a secure to an insecure state (e.g., a sequence of system calls characterizing behavior of malicious intruder [13]), or from a low-risk to a high-risk state (e.g., an unexpected load of users [7]). By providing early warning of possible, imminent risk in the dynamic execution environment, run-time monitoring is able to complement efforts to increase dependability of software via traditional testing and sound software development practices. In essence, run-time monitoring utilizes the knowledge of normal and/or abnormal system behavior and identifies problems if the system execution deviates from known normal behavior or follows a pre-specified abnormal scenario.

In this aspect, run-time monitoring resembles intrusion detection which aims at discovering malicious deviations from the expected program behavior. In the intrusion detection field existing approaches can be classified into (a)misuse-based (b) anomaly-based and (c) specification-based [18]. The misuse-based technique relies on pre-specified attack signatures, and any execution sequence matching a signature is flagged as abnormal. An anomaly-based approach, on the other hand, typically depends on normal patterns, and any deviation from normal is classified as malicious or faulty. Unlike misuse-based detection, an anomaly-based approach can detect previously unknown abnormalities. However, it relies on a machine-learning techniques which can only classify pre-specified, fixed-length behavioral patterns, and suffers from a high rate of false positives [18]. A specification-based technique operates in a similar fashion to an anomaly-based method in detecting deviations from the specified legitimate system behavior. However, in contrast to anomaly detection, a specification-based approach requires user guidance in developing a model of valid program behavior in the form of specifications. This process, though tedious and reliant on user expertise, can handle variable-length sequences and is, therefore, more accurate than an anomaly-based technique.

In this paper, we present a monitoring technique which combines the advantages of two intrusion detection approaches: anomaly-based and specification-based detection. Instead of manually developing possible, variable-length, legal behavioral patterns of a system, the approach relies on a machine-learning technique to automatically classify system behavior at runtime as *correct* or *incorrect* and infer classification of variable-length sequences.

To efficiently maintain the results of classification, we propose a novel structure *EXtended ACTion graph* (`Exact`) that appropriately combines multiple sequences classified

*The author is supported in part by NSF grants 0204139, 0205588, & 0541163. It was performed in part at the Univ. of WI.

by the machine-learning technique into variable-length patterns and memorizes them for future reference. In our framework, we have two Exact: one for storing normal patterns and the other for abnormal patterns. Sequences are classified using Exact, and the machine learning algorithm is only invoked if necessary. The following summarizes the contributions of this work:

1. *Exact structure.* Exact graph allows compact and precise representation of variable-length sequences.
2. *Automatic development of specifications.* While the machine-learning technique automatically classifies fixed-length patterns, Exact caches the results of classification in such a way that variable-length sequences can be classified in the future.
3. *Efficiency.* We describe efficient algorithms for insertion of new patterns into Exact graph and identification of existing patterns using Exact graph.

The reminder of the paper is organized as follows: a brief overview of related work is given in Section 2. An overview of the integrated framework and its components are described in Sections 3, 4 and 5. Analysis of Exact followed by the experimental results are presented in Sections 6 and 7. We conclude with the discussion of the significance of the results in Section 8.

2. Related Work

The practical benefits of the automatic detection of software errors and vulnerabilities have been noted by many researchers, and a number of techniques ranging from static program analysis [20, 10] to run-time monitoring of software executions [5, 11, 2] have been proposed over the last two decades. In the field of dynamic analysis for detecting source-code errors, Ernst et al. [5] have developed a dynamic invariant detection technique to determine fault-revealing properties of programs. Subsequent work by Hangal and Lam [11] used this technique to detect code errors by dynamically extracting invariants and checking for their violations through program execution. More rigorous work in this direction was done later by Brun et al.[2]. These approaches are close in spirit to dynamic program analysis and are specifically designed to detect errors in source-code, while our approach relies on observable system behavior.

The approach used in this paper was inspired by the technique proposed by Sekar et al. [18]. Their work aimed at augmenting machine learning techniques with high-level specifications to achieve a high degree of precision in detecting anomalies in software behavior. The authors show that the sliding window technique [9] using a machine learning algorithm may be excessively error-prone due to its inability to classify sequences of varying length. They thus manually develop high-level specifications (as finite state machines) of software systems and annotate them using information learnt via machine-learning techniques.

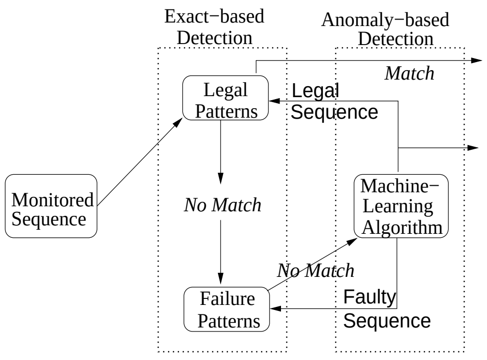


Figure 1. Framework architecture.

However, such manual development of specification is tedious and requires expert-knowledge. In contrast, we propose to generate specifications in the form of variable-length patterns automatically classified via machine learning. Specifically, we use one-class support vector machine (SVM), capable of classifying fixed-length patterns, to identify pattern-classification and infer the classification of variable-length patterns from aggregation of the results.

While the sliding window technique is a common way of modeling system data, there have been several approaches to dealing with variable-length patterns based on suffix trees [4, 15, 12, 6]. Eskin et al [6] proposed an approach for determining optimal sliding window size based on Sparse Markov Transducers (SMTs) extension of probabilistic suffix trees, that allows to consider a mixture of possible trees and estimate the best tree for the given data. While the proposed algorithm provides a good prediction for variable-length patterns in a particular data set, it is static in nature as it does not allow an update of the prediction tree as more system data becomes available.

3. Multi-Level Classifier

Our model for monitoring execution sequences of a software system consists of a two-level classification mechanism (Figure 1). Sequences in this context are defined over the observable actions performed by the system, e.g., commands issued by a controller or system calls invoked by a device driver. Specification of correct/legal and faulty behavioral patterns are provided in the first level in the form of Exact. If the sequence to be monitored matches the specifications, the second level classification is not invoked. A sequence that matches legal specifications is allowed to execute unaltered while a faulty sequence is blocked and/or appropriate actions (such as intrusion response) are fired.

If the sequence is *not* found in the specification module, the second-level classifier is used. We then rely on machine-learning techniques to determine whether the sequence is normal or anomalous. In either case, the sequence is recorded in the corresponding Exact specifications for future reference. In the domain of software reliability monitoring, a faulty behavior may result in re-visiting the requirements and correcting the implementation. In that case, the revised implementation can be monitored against prior faulty behaviors, recorded in the specification, to rule out the presence of the same errors.

One of the important features of our model is that the technique can be deployed with empty or partially filled `Exact` in the first level. As more sequences are classified by the second level, the `Exact`-level is populated automatically. In addition to acting as a cache for pre-specified classification results, `Exact` also allows future classification of patterns of any size. Also note that `Exact` is similar to low-level specifications of system behavior. In other words, the framework is generating specifications of system behavior automatically. These `Exact` specifications can also be used to retrain the second-level classifier as more new patterns become available.

4. Extended Action Graph

`Exact` is used to record previously classified behavioral patterns. In `Exact`, which is a graphical representation of multiple sequences of varying length, states are annotated by observable actions of the system to be monitored, and a sequence of states represents a behavioral pattern.

Definition 1 (`Exact`) *An Extended Action Graph E is $(S, S_0, \rightarrow, \Sigma, L)$ where S is the set of states, $S_0 \subseteq S$ is the set of start states, Σ is a set of binary numbers, $\rightarrow \subseteq S \times \Sigma \times S$ is the set of transition relations,and $L : S_0 \rightarrow \Sigma$ is a mapping of start states to a binary vector.*

A sequence in `Exact` is represented by $s_1, s_2, \ldots s_n$ where each s_i has a transition to s_{i+1}. Consider the example in Figure 2(a). The action graph, that was generated by three sequences s_1, s_2, s_3, s_2, and $s_2, s_4, s_5, s_1, s_3, s_6$, and s_1, s_3, s_6, has six states and two start states s_1 and s_2. Each transition and the start states are labeled by a binary vector; e.g., $L(s_1) = 101$. However, not all the sequences in `Exact` are classified as valid, and valid sequences form a superset of the known sequences. In the above example, s_1, s_2, s_3 and s_1, s_3, s_6 are valid patterns, and the graph also contains the sequence s_1, s_2, s_3, s_6 which is not valid.

To rule out invalidity, we use the transition label $\sigma \in \Sigma$, a binary vector, whose k-th element is denoted by $\sigma[k]$. If there exists a transition $s_i \xrightarrow{\sigma_i} s_j$ where $\sigma_i[k] = 1$, then s_i, s_j are said to be consecutive alphabets in the k-th valid sequence. Note that the first sequence is identified by setting the rightmost bit to 1, i.e., 001 is the identifier for the first sequence, 010 is the identifier for the second sequence and so on. In Figure 2(a), s_1, s_2 are consecutive states in the first pattern while s_1, s_3 are consecutive states in the second and third valid sequences. Every valid sequence is assigned a start state: s_2 is the start state of the second valid sequence. Formally, a valid sequence is defined as follows:

Definition 2 (Validity) *A sequence $s_1, s_2, \ldots, s_n$ is said to be valid if $s_1 \in S_0$, $L(s_1) = \sigma_s$ and*

$$\exists k\, \forall 1 \leq i < n : \; s_i \xrightarrow{\sigma_i} s_{i+1} \land (\sigma_s[k] = 1 \land \sigma_i[k] = 1)$$

In words, there exists a specific element in the vector-label of each transition in this sequence and the vector-identifier of the start state which is equal to 1. Furthermore, via transitivity, if $s_i, s_{i+1}, \ldots, s_{i+n}$ is a valid sequence and $s_j, s_{j+1}, \ldots, s_{j+m}$ is another valid sequence such that $s_{i+n} = s_j$ then $s_i, s_{i+1}, \ldots, s_{(i+n)-1}, s_j, s_{j+1}, \ldots, s_{j+m}$ is also a valid sequence.

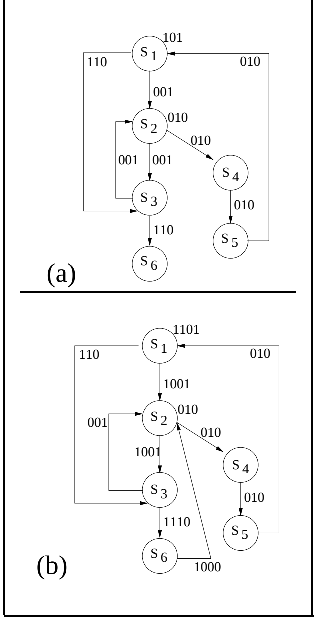
Validity takes care of unbounded (one or more) repetition of the alphabets in a sequence, e.g., in Figure 2(a) $s_1, s_2, s_3, s_2, s_3, \ldots$ is a valid sequence. In the above, s_2 is said to be the root of the loop. A sequence representing *exit* from a loop is classified as a *new* sequence starting from the root of the loop. In Figure 2(a), $s_2, s_4, s_5, \ldots$ is the valid sequence exiting from the loop rooted at s_2. Note that the transitivity relation in Definition 2 can be used to identify valid sequences with bounded repetition (from a valid sequence with finite looping and a valid exit sequence). For example in Figure 2(a), s_1, s_2, s_3, s_2 and s_2, s_4, s_5 are valid sequences and they form, via transitivity, a new valid sequence $s_1, s_2, s_3, s_2, s_4, s_5$.

Searching and constructing `Exact` Figure 2(c) presents the algorithm to find out whether a given sequence is a valid sequence in `Exact`. Procedure `match` takes as input the given sequence $s_{l\ldots k}$, a bit-vector `m` and the `Exact` and returns true if the sequence is a valid sequence in `Exact`. `Exact` is deterministic, i.e., for every pair of states there exists at most one transition. Absence of non-determinism makes the complexity of searching for a valid sequence linear in the size of the given sequence and in terms of `Exact` size, the worst case complexity is $O(n \times s)$ where n is the sequence length and s is the number of states in `Exact`.

Figure 2(d) presents the algorithm for insertion of a new sequence in `Exact` graph. Procedure `insert` takes as arguments the sequence to be inserted, a bit vector `m` identifying the new sequence and the graph `Exact`. The procedure `match` is always invoked before inserting any new sequence to avoid duplicate insertions. As such, the worst case complexity is $O(n \times s)$ where n is the sequence length and s is the number of states in `Exact`.

Illustrative Example Let $s_1, s_2, s_3, s_6, s_2, s_4, s_5$ be a sequence to be inserted in the example `Exact` in Figure 2(a). First we break the sequence up into substrings s_1, s_2, s_3, s_6, s_2 (up to the first repeated alphabet s_2) and s_2, s_4, s_5 (following the transitivity rule). Then we insert these two subsequences following these steps:

1. This sequence s_1, s_2, s_3, s_6, s_2 is not present in the `Exact` graph shown in Figure 2(a) as there is no



```
bool match($s_{l\ldots k}$, m, E) {
 if ($s_l \in S_0$) && ($s_l \overset{\sigma_l}{\rightarrow} s_{l+1}$) {
  if (m=m & $\sigma_l$ & $L(s_l)$) && (m!=0) {
   int i=l+1; visited($s_l$) = true;
   while (i<k) {
    if ($s_i \overset{\sigma_i}{\rightarrow} s_{i+1}$) {
     if (m=m & $\sigma_i$)&&(m!=0) {
      i=i+1; visited($s_i$)=true; }
     else if visited($s_i$) {
       reset(visited); m=set(1);
       return match($s_{i\ldots k}$, m, E);
     }
    } else return false;
   } // end-while
   return true;
  } else return false;
 } else return false;
}
```

(c)

```
void insert($s_{l\ldots k}$, m, E) {
 int m1=set(1); // m1 is all 1's
 if match($s_{l\ldots k}$, m1, E) return;
 make_start($s_l$, $S_0$); update_L($s_l$, L, m);
 int i=l;
 while (i¡k) {
   if (visited($s_i$)) {
    reset(visited); m=m¡¡1;
    insert($s_{i\ldots k}$, m, E);
     return;
   } // end of if-then
    else {
     visited($s_i$)=true;
     update($s_i$, $s_{i+1}$, m, E);
     i=i+1;
   } // end of if-else
} // end of while
 return;
}

void update($s_i$, $s_j$, m, E) {
 if ($s_i \overset{\sigma}{\rightarrow} s_j$ ∈E) $\sigma = \sigma \mid$ m;
 else connect($s_i \overset{m}{\rightarrow} s_j$, E);
}
```

(d)

Figure 2. (a) Example of an `Exact` graph. (b) `Exact` graph in Figure 2(a) shown after insertion of $s_1, s_2, s_3, s_6, s_2, s_4, s_5$. (c) Pseudo-code for `Exact` search. (d) Pseudo-code for `Exact` insert.

transition from s_6 to s_2. The `match` algorithm (Figure 2(c)), in this case, makes an *early* detection of its absence as the prefix s_1, s_2, s_3, s_6 of the given sequence is not valid in the exact. Observe that, for this prefix, bit-wise "and"-ing of the transition labels and the start state label results in 0 (101 `AND` 001 `AND` 001 `AND` 110 = 000, see Figure 2(a)) and our `match` algorithm (Figure 2(c)) returns false. As such, s_1, s_2, s_3, s_6, s_2 is inserted as a new sequence. Further, as s_1 is already present in the set of start states, its start-state label is updated using the new sequence identifier 1000. Recall that the identifier for the first sequence is 001, identifier for the second sequence is 010 and for the third sequence is 100. As such the identifier for the new (fourth) sequence is 1000.

2. Each transition of the new sequence is added to `Exact` graph with identifier 1000. We start with transition $s_1 \rightarrow s_2$. It already exists in the graph and its identifier is 001. Applying *bitwise-OR* of the existing and new transition label we obtain 1001 and update the transition with this new label (Figure 2(b)). We continue in this fashion until we reach transition substring s_6, s_2. There is no transition between s_6 and s_2, therefore, a new transition ($s_6 \rightarrow s_2$), is added with the label 1000.
3. Due to the repeated appearance of s_2, the second sequence s_2, s_4, s_5 is set up to be inserted as a new sequence with a new sequence number, 10000. However, its insertion is avoided as the sequence s_2, s_4, s_5 is already a valid sequence in `Exact`. The updated `Exact` graph is shown in Figure 2(b).

5. Second-level Classifier

Any machine learning technique can be applied as a second-level classifier in our framework. For our case study, we used one-class SVM which allows usage of *unlabeled data*. As opposed to its classical version, two-class SVM [19], one-class SVM relies on maximally separating all data from origin using a hyperplane [17].

The unlabeled data, in our case, are sequences of observable actions representing the system behavior we are interested in monitoring. Observability may be defined in different ways in different application domains; for example, in-flight stability or collision avoidance controllers, we use the SVM to classify *pairs of input and output control signals* [14], while in host-based software intrusion detection, we are interested in classifying *sequences of system calls*.

For the purpose of discussion, we illustrate the application of SVM classifier via an example. Let the observed input stream be `Istream` $\equiv s_1, s_2, s_3, s_2, s_3, s_4, s_2$ and `Exact` in Figure 2(b) failed to recognize `Istream` as a valid sequence. First, we break-up `Istream` following the transitivity relationship in Definition 2 of Section 4, i.e., $\texttt{Seq}_1 \equiv s_1, s_2, s_3, s_2$ and $\texttt{Seq}_2 \equiv s_2, s_3, s_4, s_2$. Note that the break-up point is at s_2 which appears in $\texttt{Seq}_1$ and $\texttt{Seq}_2$, and is the first alphabet to be repeated in `Istream`. SVM can only take fixed length sequences as input and as such we apply classic sliding window technique to provide inputs to the SVM. Let the sliding window size be 3, then

W = 7, L=4 Exact sequences: s_1, s_2, s_3, s_4 & s_4, s_5, s_6, s_7 Combined sequence: $s_1, s_2, s_3, s_4, s_5, s_6, s_7$ SVM sequence: $s_1, s_2, s_3, s_4, s_5, s_6, s_7$ (a) `Exact vs Backend-SVM:` $W > L$	W = 3, L=5 Exact sequence: s_1, s_2, s_3, s_4, s_5 Combined sequence: s_1, s_2, s_3, s_4, s_5 SVM sequences: s_1, s_2, s_3 & s_2, s_3, s_4 & s_3, s_4, s_5 (c) `Exact vs Backend-SVM:` $W < L$
W = 4, L=3 Exact sequences: s_1, s_2, s_3 & s_3, s_4, s_5 Combined sequence: s_1, s_2, s_3, s_4, s_5 SVM sequences: s_1, s_2, s_3, s_4 & s_2, s_3, s_4, s_5 (b) `Exact vs Backend-SVM:` $W > L$	W = 6, L=3 Incoming sequence: $s_1, s_2, s_3, s_4, s_5, s_6, s_7$ Exact sequences: s_1, s_2, s_3 & s_3, s_4, s_5 & s_5, s_6, s_7 SVM sequences: $s_1, s_2, s_3, s_4, s_5, s_6$ & $s_2, s_3, s_4, s_5, s_6, s_7$ (d) `Our framework vs Backend-SVM`

Figure 3. Examples

SVM is fed with subsequences: (i) s_1, s_2, s_3, (ii) s_2, s_3, s_2 (from $\texttt{Seq}_1$), (iii) s_2, s_3, s_4 and (iv) s_3, s_4, s_2 (from $\texttt{Seq}_2$). Finally, $\texttt{Seq}_1$ and $\texttt{Seq}_2$ are termed as normal iff all their subsequences are classified by SVM as normal. Note that, break-up of $\texttt{Seq}_1$ and $\texttt{Seq}_2$ using sliding window does not adversely effect end result, i.e., if any subsequence of $\texttt{Seq}_1$ or $\texttt{Seq}_2$ is classified as anomalous, then the corresponding sequence is conservatively classified as anomalous.

6. Analysis of Exact

As `Exact` represents variable-length sequences, the comparison with models based on the fixed-length patterns is challenging, with the main challenge being the difference in the number of fixed-length and variable-length sequences generated from the same data set. The comparison is also aggravated by the fact that classification of variable-length patterns in `Exact` depends entirely on the underlying fixed-length classifier (SVM in this case).

In this section, we present a comparative study of the number of sequences being examined by `Exact` and the SVM classifier. We consider two possible cases: one where the SVM, used in conjunction with `Exact`, acts as the backend for our framework (backend SVM) and the other where SVM acts alone (stand-alone SVM). The comparison will form the basis for results presented in the Section 7.

For the purpose of analysis, we will consider the average length of `Exact` sequences L which computed using the weighted mean where the weight of a length denotes the number (frequency) of sequences of the corresponding length. Let the fixed sliding window size of SVM be W.

Exact vs Backend-SVM. The two possible scenarios of interest are $W > L$ and $W < L$. For $W = L$, the number of `Exact` sequence and SVM sequence is identical.

1. $W > L$: In this case, several `Exact` sequences are combined to form one SVM sequence (example in Figure 3(a)). Consider first the case where x `Exact` sequences fit *exactly* in one SVM sequence of size W. In other words, $xL - (x-1) = W$ (the subtraction of $x - 1$ from xL is required to account for overlap between two consecutive `Exact` sequences). Therefore, $x = \frac{W-1}{L-1}$. In other words, the number of `Exact` sequences is greater than the number of SVM sequences, and classification of one SVM sequence influences the classification of x `Exact` sequences.
   Secondly, consider the case where $\frac{W-1}{L-1}$ is not a whole number, i.e. the `Exact` sequences is not a integer-multiple of SVM sequences. Let x be the smallest number of `Exact` sequences such that $xL - (x-1) > W$ and $\forall y < x : yL - (y-1) < W$. Therefore, the number of SVM sequences corresponding to x `Exact` sequences is, $xL - (x-1) - W + 1 = x(L-1) - W + 2$. Then the number of `Exact` sequences is greater than the number of SVM sequences if $x < \frac{W-2}{L-2}$; otherwise the number of SVM sequences is greater or equal to the number of `Exact` sequences (example in Figure 3(b)). In the case of the former, one SVM sequence classification influences one `Exact` sequence classification while in latter, one SVM sequence can potentially effect x `Exact` sequences.
2. $W < L$: The number of `Exact` sequences is less than the number of SVM sequences (example in Figure 3(c)). Specifically, the number of SVM sequences corresponding to one `Exact` sequence is $(L - W + 1)$ and therefore, one SVM sequence classification can effect the classification of one `Exact` sequence.

Our framework vs. Stand-alone SVM. Next, we consider the number of sequences examined by SVM if it is deployed alone. Given that the total length of the input stream is IS, the total number of SVM sequences is $N = IS - W + 1$. If the same input stream is input to our framework – `Exact` with backend-SVM – the total number of `Exact` sequences is $(IS - 1)/(L - 1)$, i.e. $(N + W - 2)/(L - 1)$. The number of sequences examined by SVM alone is greater than the number of `Exact` sequences in our framework if $N > (W - 2)/(L - 2)$ (example in Figure 3(d)).

Also, note that if $W < L$, the number of sequences examined by SVM, when deployed alone, can be potentially greater than the number of sequences examined by SVM, when deployed in conjunction with `Exact`. Specifically, the situation requires $N > L - W + 1$ and can be explained from the fact that number of sequences classified by backend-SVM depends on the number of `Exact` sequences when $W < L$.

Number of	stand-alone SVM			Exact (variable-length sequences)		
	snsndmailcp	decode	fwdloop	snsndmailcp	decode	fwdloop
Normal sequences in train data set	30792	30792	30792	3314	3314	3314
Sequences in test data set	1098	2983	2499	78	405	204
Anomalous sequences in test data set	264	741	387	24	92	43

Table 1. Information on sendmail normal and intrusive trace data sets

	snsndmailcp	decode	fwdloop
Exact of normal specifications	5	16	13
Exact of faulty specifications	14	31	39

Table 2. Max. length of Exact binary vectors

7. Case Study

We evaluated our model in the security domain using synthetic sendmail data provided by the UNM [8]. Sendmail data is an unlabeled collection of system calls. It consists of a normal data sets which contain only legal patterns and trace data sets containing normal patterns as well as anomalies. We considered three intrusion trace data sets: snsndmailcp, decode and fwdloops. The one-class SVM classifier was trained on the normal data set (training set), tested on the trace sets (test sets) and implemented using libsvm tool [3] and the window size of 8.

Data Sets. Table 1 presents the pattern of data being used for evaluation pupose in terms of number of sequences. The training data set contains 30792 normal fixed-length sequences. On the other hand, using Exact, the number of variable-length sequences is 3314. The decrease in the number of sequences is due to the fact that Exact partitions sequences using repetitions and as such can handle variable-length sequences (see Section 4). We then processed the normal and abnormal patterns of the test data set to generate two test sets: one for stand-alone SVM, containing fixed-length sequences obtained through sliding window technique, and one for Exact, containing variable-length sequences generated in Exact fashion (row 2 in Table 1). Finally, the last row shows the number of sequences that are in the test data set but are not present in the training data set. For example, out of 1098 fixed-length sequences for snsndmailcp, there are 264 sequences which are not present in fixed-length sequences of training data. For the purpose of evaluation, we can conservatively assume that sequences not present in the training data set are anomalous; the goal is to identify all such anomalous sequences.

Table 2 presents the maximum length of binary vectors after building Exact graphs on data sets, i.e., the number of distinct variable-length sequences in Exact.

Efficiency. In these experiments we focused primarily on the rate of populating the Exact with normal(legal) and abnormal(anomalous) patterns. To evaluate our technique we monitored the stage at which each sequence was classified. We examined two scenarios:

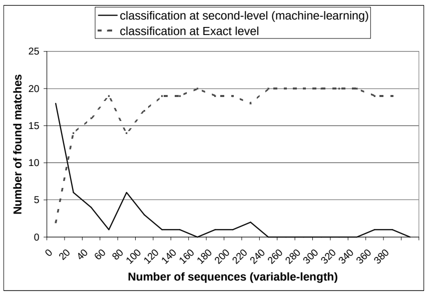


Figure 4. Initialization: empty Exact

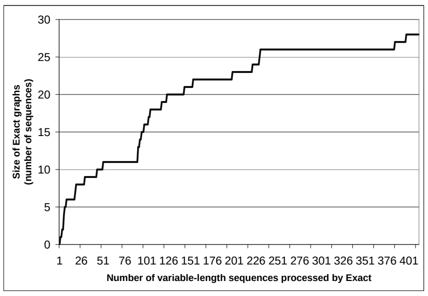


Figure 5. Populating Exact

1. Both Exact graphs representing normal and abnormal specifications are initially empty
2. Partial specification is available initially, i.e., the Exact graph corresponding to a normal specification is populated with 10% of the patterns from the normal data set.

The results for both scenarios corresponding to *decode* intrusion are presented in Figures 4, 5 and 6.

Figure 4 shows the frequency at which both levels of classifiers (Exact and backend SVM) were invoked for classifying the incoming sequences. Since simulation started with an empty Exact graph, almost every incoming sequence is classified at the second-level classifier. However, the access rate of second-level classifier rapidly decreases as more patterns were stored in the Exact. Consequently, the number of sequences classified at the Exact level increases. Figure 5 shows the number of new patterns added to the empty Exact over the same run of decode trace set. The majority of patterns were recorded within about 200 sequences (out of 405 total sequences). After that, almost all patterns were found at the Exact level.

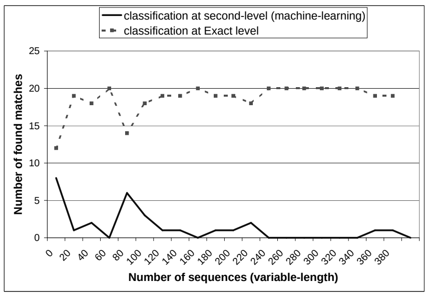


Figure 6. Initialization: `Exact` partially populated with 10% of normal sequences

The result corresponding to the second scenario where the normal `Exact` graph is partially populated is shown in Figure 6. As opposed to the previous figure, the access rate of the second-level classifier in the beginning of the run is low while the `Exact` graph access rate is high. This is explained by the partial presence of the sequences in the normal `Exact` specifications. However, since only partial normal patterns were added to the specifications, the second-level classifier was still accessed whenever new normal or anomalous sequence is found. In this scenario we benefited from the available specifications having populated the `Exact` in advance. This shortened the start-up time necessary to store a sufficient number of patterns (Table 3)[1]. The processing time for 405 sequences was 2 times faster with the populated specifications (7 sec) than with the empty specifications (16 sec). Also, it is the SVM classifier access that requires most of this time.

Accuracy. As the `Exact` graph provides a succinct representation of learned through machine-learning technique variable-length patterns, we focused in these experiments on the comparison of the accuracy of our structure to the accuracy of SVM tested on a model built using the sliding window technique. For evaluation purpose we considered *detection rate* (ratio of detected anomalies to the total number of anomalies presented in the set) and *false positive rate* (FP) (normal instances incorrectly identified as anomalous).

As Table 4 show, classification results of fixed-length patterns for stand-alone SVM and `Exact` integrated with SVM are similar. For example, for `snsndmailcp`, the detection rate is 98% for both stand-alone SVM and backend SVM used in `Exact`. The results confirm the fact that `Exact` structure, while recording variable-length patterns, truly represents information given by the backend classifier in compact fashion. The existing 1–2% variation in the results is explained by the potential difference in the number of sequences examined by stand-alone SVM and back-end SVM used in `Exact` (see Section 6).

The classification results are also given in terms of variable-length patterns stored by `Exact` (Table 5). Examining it, we notice that prediction results are slightly different from the corresponding percentanges given in Table 4. For example, the detection rate of `Exact` for snsndmailcp intrusion given in fixed-length patterns is 98% while the corresponding number of detected variable-length sequences is 24 out of 24. This happens when several SVM sequences, including those that are correctly classified as anomalous and those that represent missed intrusions, are combined into one `Exact` sequence resulting in an anomalous `Exact` sequence and providing a higher detection rate.

[1]Average over 10 runs.

	Total time(sec)	Backend SVM running time(sec)
`Exact` with empty specs	16	12
`Exact` with partially populated specs	7	5

Table 3. Running time(*decode* intrusion).

An opposite scenario is represented by `decode` intrusion, where the detection rate in fixed-length patterns is 100% which corresponds to 90 out 92 variable-length `Exact` sequences. Closer inspection reveals that the result is as expected and can be explained by the fact that `Exact` records sequences depending on the classification result from backend SVM classifier. There are a couple of occurrences of one particular `Exact` sequence in the test data set which is not present in the training data set. Hence, this sequence is classified as an anomaly (counted as one of the anomalous patterns among 92 anomalies: see Table 1). It turns out that the length of the sequence is 2 due to two consecutive identical system call-invocations. As such the SVM using sliding window size 8 does not consider this sequence independently; instead it combines the sequence with another (next) `Exact` sequence and performs classification. As the combined sequences are classified as normal by SVM, the `Exact` also records the combined sequence as normal. This is acceptable as the main purpose of `Exact` is to memorize variable length sequence and closely follow SVM classifier. Note that if the SVM classifier used window size of 2, then the above scenario would be removed.

The number of variable-length sequences falsely recognized as positive in `Exact` is also different from the corresponding percentage given for fixed-length sequences. This is due to the fact that several SVM sequences can represent one `Exact` sequence, thus significantly reducing the total number of variable-length sequences in comparison to those in fixed-length. The detailed analysis of this dichotomy was presented in Section 6. At the same time, manual inspection of these results showed that a number of FP sequences in `Exact` fully comes from the backend SVM.

While the trade-off between the number of detected and the false positives is inherently present in many machine learning algorithms including SVM, this error can be effectively reduced with guidance from normal specifications. In fact, populating `Exact` even with the small number of nor-

	stand-alone SVM			Our framework (results from the backend SVM based on fixed-length seqs)		
	snsndmailcp	decode	fwdloop	snsndmailcp	decode	fwdloop
Detection rate	98%	99%	99%	98%	100%	100%
FP rate	11%	7%	10.7%	13%	8%	10%

Table 4. Accuracy of classification with empty `Exact` shown in fixed-length sequences.

	empty `Exact`			`Exact` populated with 10% of normal sequences		
	snsndmailcp	decode	fwdloop	snsndmailcp	decode	fwdloop
Number of detected sequences	24 out of 24	90 out of 92	42 out of 43	24	90	42
FP sequences	21 out of 54	62 out of 313	75 out of 161	0	1	9

Table 5. Accuracy of our framework classification shown in variable-length sequences.

mal patterns reduces the number of false positives significantly (Table 5). Since the overall variability of sendmail behavior is small, even approximately 10% of normal sequences leads to recognition of majority of normal patterns. However, generally a greater variability in process behavior might require a larger set of normal patterns to improve the accuracy of classification. Note that an `Exact` with partially populated normal specification does not affect the detection rate. This is because abnormal, incoming sequences are still recognized as unknown and processed by SVM as they would be if the `Exact` graphs were empty.

8. Conclusion

We present fast online classification of run-time behavioral patterns for software systems based on the combination of specification and anomaly-based approaches. We show that memorization of classification results from the SVM classifier can be effectively applied to generate (partial) specifications automatically. We introduce the data structure `Exact` for recording specifications and develop efficient algorithms for insertion and matching of sequences. Finally, our experimental results indicate that our technique can be effectively applied in practical setting in the domain of intrusion detection.

Recently, Bowring et al. [1] proposed a technique to predict program behavior by modeling program executions as Markov models and merge similar/redundant models using clustering. In future work, we plan to investigate the effect of incorporating Bowring's approach into the generation of `Exact` graph, specifically by annotating stochastic information with each `Exact` transition.

Another important avenue of future research is the classification of found faults according to their severity. This can help to explain the cause of an abnormality once it is detected using `Exact` specifications. We also plan to enhance the efficiency of our approach by combining the `Exact` graphs for normal and faulty specifications to support *early* identification of sequences as normal or abnormal.

References

[1] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. *SIGSOFT Softw. Eng. Notes*, 29(4), 2004.

[2] Y. Brun and M. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE*, 2004.

[3] C.-C. Chang and C.-J. Lin. Libsvm: a library for support vector machines (version 2.31). Available from ”http://www.csie.ntu.edu.tw/c̃jlin/libsvm/”.

[4] H. Debar, M. Dacier, M. Nassehi, and A. Wespi. Fixed vs. variable-length patterns for detecting suspicious process behavior. In *ESORICS '98*, 1998.

[5] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *IEEE TSE*, volume 27, 2001.

[6] E. Eskin, W. Lee, and S. J. Stolfo. Modeling system calls for intrusion detection with dynamic window sizes. In *DISCEX II*, 2001.

[7] S. Fickas and M. S. Feather. Requirements monitoring in dynamic environments. In *RE*, 1995.

[8] S. Forrest. Computer immune systems, data sets. Available from ”http://www.cs.unm.edu/ĩmmsec/data/synth-sm.html”.

[9] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *SP'96*, 1996.

[10] R. Gopalakrishna, E. H. Spafford, and J. Vitek. Efficient intrusion detection using automaton inlining. In *SP '05*, 2005.

[11] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02*, 2002.

[12] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Software*, 14:24–42, 1997.

[13] Y. Liao and V. R. Vemuri. Use of k-nearest neighbor classifier for intrusion detection. *Computers & Security*, 21(5):439–448, 2002.

[14] Y. Liu, S. Yerramalla, E. Fuller, B. Cukic, and S. Gururajan. Adaptive control software: Can we guarantee safety? In *COMPSAC*, pages 100–103, 2004.

[15] C. Marceau. Characterizing the behavior of a program using multiple-length n-grams. In *NSPW*, pages 101–110, 2000.

[16] A. Mili, G. Jiang, B. Cukic, Y. Liu, and R. Ayed. Towards the verification and validation of online learning systems: General framework and applications. In *HICSS*, 2004.

[17] B. Scholkopf, J. Platt, J. Shawe-Taylor, A. Smola, and R. Williamson. Estimating the support of a high-dimensional distribution. Technical Report 99-87, Microsoft Research, 1999.

[18] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang, and S. Zhou. Specification-based anomaly detection: a new approach for detecting network intrusions. In *CCS '02*, 2002.

[19] V. Vapnik. *Statistical Learning Theory.* Wiley, New York, 1998.

[20] D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01*, 2001.