On the symbiosis of specification-based and anomaly-based detection

Natalia Stakhanova\textsuperscript{a,}\textsuperscript{*}, Samik Basu\textsuperscript{b}, Johnny Wong\textsuperscript{b}

\textsuperscript{a}Faculty of Computer Science, University of New Brunswick Fredericton, NB E3B 5A3, Canada \\
\textsuperscript{b}Department of Computer Science, Iowa State University Ames, IA 50011, USA

Abstract

As the number of attacks on computer systems increases and become more sophisticated, there is an obvious need for intrusion detection systems to be able to effectively recognize the known attacks and adapt to novel threats. The specification-based intrusion detection has been long considered as a promising solution that integrates the characteristics of ideal intrusion detection system: the accuracy of detection and ability to recognize novel attacks. However, one of the main challenges of applying this technique in practice is its dependence on the user guidance in developing the specification of normal system behavior. In this work, we present an approach for automatic generation of specifications for any software systems executing on a single host based on the combination of two techniques: specification-based and anomaly-based approaches. The proposed technique allows automatic development of the normal and abnormal behavioral specifications in a form of variable-length patterns classified via anomaly-based approach. Specifically, we use machine-learning algorithm to classify fixed-length patterns generated via sliding window technique to infer the classification of variable-length patterns from the aggregation of the machine learning based classification results. We describe the design and implementation of our technique and show its practical applicability in the domain of security monitoring through simulation and experiments.

1. Introduction

The rapid increase in the number, sophistication and impact of computer attacks makes the computer systems unpredictable and unreliable, emphasizing the importance of intrusion detection ability to correctly recognize known attacks and identify new threats.

Typically, intrusion detection refers to a variety of techniques for detecting attacks in the form of malicious and unauthorized activities. There are three broad categories of detection approaches (Sekar et al., 2002) (a) misuse-based (b) anomaly-based and (c) specification-based. Misuse-based technique relies on pre-specified attack signatures, and any execution sequence matching with a signature is flagged as abnormal. An anomaly-based approach, on the other hand, depends on automatic classification of executions as normal patterns, and any deviation from normal patterns is classified as malicious or faulty. Unlike misuse-based detection, anomaly-based techniques can detect previously unknown abnormalities. However, anomaly-based approaches rely on statistical or machine learning classification techniques which can only classify (usually) pre-specified, fixed-length behavioral patterns, and suffer from the disadvantage of a high rate of false positives (Lazarevich et al., 2003). Specification-based techniques operate in a similar fashion to anomaly-based method detecting deviations from the...
specified legitimate system behavior. However, as opposed to anomaly-based detection mechanism, specification-based approach requires user guidance in developing model of valid program behavior in a form of specifications. This process can handle variable-length sequences and is more accurate than anomaly-based techniques, but it can be prohibitively tedious and error-prone due to reliance on level of user-expertise.

1.1. Problem statement

The research in intrusion detection field has been mostly focused on anomaly-based and misuse-based detection techniques for a long time. This is due to the ability of anomaly detection approach to recognize novel attacks and predictability of misuse-based detection models (Innella, 2001; Kemmerer and Vigna, 2002). Although the specification-based detection has been less favored due to the inherent difficulty of developing specifications, it effectively combines the advantages of the other two approaches: the accuracy and the ability to detect new attacks, while avoiding their shortcomings. Sekar et al. (2002) emphasized another advantage of specification-based approach, its ability to accommodate the variable-length patterns that naturally represent the system behavior. (Sekar et al., 2001) showed that the traditional anomaly detection models employing machine learning-based n-gram techniques may be error-prone in certain cases due to inability to classify patterns of variable-length.

Although the strengths of specification-based approach are obvious, the question of these benefits’ applicability in practice still remains open. One of the main challenges in this context is the amount of effort required to develop normal behavioral specifications of large systems.

1.2. Solution methodology

In this paper, we explore the benefits of the specification-based approach in the security domain. Specifically, we analyze the capability of variable-length patterns to effectively represent system behavior and consequently, serve as a basis for system behavior specification.

In this work, we present an adaptive technique for automatic online learning and detection of abnormal program behavior through combination of two intrusion detection approaches: anomaly-based and specification-based detection. Such combination recognizes a known behavior of the system through the specifications of normal and abnormal patterns while classifying unknown behavior using a machine-learning algorithm.

Furthermore, in the absence of a pre-specified specification, the proposed approach can also be applied to automatically develop specifications using the classification results of machine-learning algorithms. Instead of manually developing all possible variable-length patterns of a legal system behavior, we use the machine-learning algorithm to classify fixed-length patterns as normal and/or abnormal, and then appropriately combine these classified patterns to create normal or abnormal behavioral specifications of variable-length. This technique of obtaining feedback from the machine-learning algorithm to create specification can also be used to augment and adapt pre-existing possibly incomplete specifications of normal behavior.

To efficiently maintain the results of classification and detect variable-length known/classified patterns, we propose a novel data structure EXtended ACTion graph (Exact). Exact was initially introduced in (Stakhanova et al., 2006) for automatic caching of monitored patterns in IDS. This work extends our previous research and provides a detailed study of Exact algorithms along with theoretical analysis between Exact patterns and fixed-length patterns. Furthermore, we also present a prototype showing the practical applicability of Exact in IDS.

In essence, the structure stores the specification of the normal and abnormal behavior of the system. Exact appropriately combines multiple sequences classified by a machine-learning technique into variable-length patterns and memorizes them for future reference. In our framework, we have two Exact structures: one for storing normal patterns and the other for abnormal patterns. Monitored sequences are classified using Exact, and the machine-learning algorithm is only invoked if necessary, i.e., if the monitored sequence does not match with any patterns stored in Exact. The following summarizes the contributions of this work:

1. Adaptive detection of anomalies. We employ a machine learning approach to classify unknown program behavior and memorize it for future reference, thus, effectively allowing the intrusion detection system to adapt to previously unknown normal and abnormal behaviors.
2. Automatic development of specifications. While the machine-learning technique automatically classifies fixed-length patterns, Exact caches the results of classification as variable-length sequences.
3. Novel specification of behavior. Exact allows compact and precise representation of variable-length sequences as specifications of normal and abnormal behavior.
4. Efficiency of our approach. We describe efficient algorithms for insertion of new patterns into Exact graph and identification of existing patterns using Exact graph. We present a prototype implementation of our technique and provide simulation results showing its effective applicability in the setting of system call based intrusion detection.

1.3. Organization

The remainder of the paper is organized as follows. A brief overview of related work is given in Section 2. Section 3 presents the Exact structure. Section 4 discusses the design and prototype implementation of our intrusion detection framework. Experimental results based on simulation and system prototype are given in Section 5. Section 6 concludes the paper with the discussion of future work.

2. Related work

The benefits of the specification-based approach have been noted by many researchers. The first specification-based models were introduced by Ko et al. (Ko et al., 1994, 1997, 2001;
properties of the program specification. However, the behavior. Essentially, this policy can be viewed as security policy during the program's execution as a list of predicates. Provos (2003) introduced an interactive generation of the code availability which is not always a realistic requirement. Control-flow analysis of source code, the approach assumes automatically derive program specifications. Based on background knowledge. While this background knowledge is emphasized another advantage of specification-based approach; its ability to accommodate the variable-length patterns that naturally represent the system behavior. This aspect is especially important as fixed-length n-gram approach (Forrest et al., 1996) may lead to inaccurate classification (Sekar et al., 2001). This inaccuracy primarily stems from the lack of generalization and ineffectiveness of the approach in capturing correlations among system calls that occur over longer time spans. To account for this limitation, Sekar et al. (2002) proposed to manually develop high-level specifications of network protocols as finite state machines (FSM) and annotate them using statistical information learned via traffic monitoring. Using this approach, the behavior of protocol is effectively represented as state machine transitions. To help differentiate between usual and abnormal behavior, each transition of FSM is further augmented with statistical properties of the network traces, specifically, distribution of network packets, the most common value and distribution of values. Motivated by this idea, we propose to generate specifications for any software system in host-based domain automatically in the form of variable-length patterns classified via machine learning. Specifically, we use machine-learning algorithm to classify fixed-length patterns generated via sliding window technique to infer the classification of variable-length patterns from the aggregation of the machine learning based classification results.

One of the major downsides of the specification-based approach is the necessity to develop the system specifications manually, which is not only a time-consuming and error-prone process, but also requires a considerable expert knowledge. Thus, the primary challenge in this domain is the automatic generation of the specification.

As such, Ko (2000) presented a semi-automatic approach for developing security specifications. He proposed a technique based on Inductive Logic Programming (ILP) method to generate program behavioral specifications represented in the form of fixed-length patterns of system calls. To ensure validity of developed specifications, each system call representing valid behavior is accompanied by pre-specified background knowledge. While this background knowledge allows to produce concise and intelligent specification understandable by humans, it also calls for substantial expert work required to generate this knowledge for each system call for a given program.

Wagner and Dean (2001) employed static analysis to automatically derive program specifications. Based on control-flow analysis of source code, the approach assumes code availability which is not always a realistic requirement. Provos (2003) introduced an interactive generation of the policy during the program’s execution as a list of predicates describing the allowed and denied actions in the program behavior. Essentially, this policy can be viewed as security properties of the program specification. However, the generated predicates are meant to provide only a base policy, and thus, require further review by the user.

The common problems in constructing the specifications automatically are a significant overhead caused to the system by the complexity of the proposed approach and a mandatory access of the specification-generating technique to the program source code. In contrast, we propose to automatically generate specifications from the observable program behavior without relying on access to the application source code. To achieve a high degree of precision in differentiating normal and anomalous behavior of the program we integrate development of specifications with a machine-learning technique. Specifically, we generate specifications automatically in the form of variable-length patterns classified via machine learning.

One of the main challenges in this context is the efficient matching of variable-length patterns. The question of storing variable-length patterns to allow matching and retrieval of patterns with low time and space complexity has been studied by many researchers in various fields.

In intrusion detection there have been several studies focused on efficient pattern matching for network packets classification (Antonatos et al., 2005; Dharmapurikar and Lockwood, 2006; Song and Lockwood, 2005; Sourdies et al., 2006). Among general approaches to variable-length sequences is the Trie structure (Fredkin, 1960) that stores patterns with common prefixes in the common node. Due to high redundancy of stored patterns and consequently, high space requirements, many improved variations of the Trie were developed.

One of such variations, the suffix tree, found a wide application in sequence matching as it allows to effectively combine all common suffixes of the pattern. Debar et al. (1998) proposed generation of variable-length patterns based on suffix trees augmented with a frequency of each subsequence. Although the approach showed the advantages of variable-length patterns, the experiments revealed no clear advantage over the traditional fixed-length approaches. Similarly, Marcelo (2000) employed a suffix tree as the underlying structure for constructing finite state machine where states represent predictive sequences of variable-length. Kosoresow and Hofmeyr (1997) manually constructed finite automaton based on variable-length patterns and applied it for detection process. Since the automaton was generated manually, the approach might not be applicable to large systems. While suffix trees provide many advantages over the Trie structure, they generally require significant storage space and construction time if long sequences are introduced.

Wespi et al. (2000) introduced an alternative approach to variable-length pattern generation. It is based on Teiresias (Rigoutsos and Floratos, 1998), an algorithm initially introduced for discovering patterns in biological sequences. While the original Teiresias algorithm extracts maximum length sequences from the data set, the approach in Wespi et al. (2000) takes a step further aggregating the consecutive elements of the obtained maximum length sequences and removing the repetitive or unused patterns. It is intended for offline construction of variable-length patterns set based on all correct process executions. While the availability of data might be a valid assumption in certain cases (e.g., generation
of legal executions through functionality verification tests (FVT)), it generally restricts the applicability of the approach. It also affects the ability of the approach to accommodate new patterns, essentially requiring to re-process the whole data set. Another limitation of this approach is its insensitivity to the sequential nature of event streams. The pattern extraction and further aggregation does not retain information about order of the patterns seen in the process. This potentially opens a vulnerability window allowing attackers to craft intrusive sequences using the available patterns.

In summary, these algorithms focus on space and time efficiency of the pattern matching and storing. However, in the context of intrusion detection systems several other issues need to be addressed such as (a) ability of the algorithm to precisely record the patterns and (b) dynamically update the structure as more data becomes available, (c) capability of the structure to incorporate patterns of various length and, finally, (d) minimal dependence of the algorithm on the user guidance. To address these issues, we introduce the Exact structure that is specifically designed to accommodate patterns of variable-length, illuminating the ineffectiveness of n-gram approach. We propose a combination of specification-based and anomaly-based detection techniques to allow automatic generation of program specifications and dynamic adaptation of generated profiles to novel behavior. Finally, we limit manual supervision throughout the training and detection phases through seamless integration of Exact structure and detection techniques.

3. Adaptive intrusion detection framework

In general, behavior of software systems can be specified in two ways (Uppuluri, 2003): by internal system state, e.g., using value of the variables, or through observable interactions between the system and the environment, e.g., using commands issued by a controller or system calls invoked by a device driver.

The main challenge of monitoring internal system state is the necessity to equip the system with internal monitors which often requires program-rewriting and, thus, can be an expensive approach. In the latter approach, program behavior can be captured by monitoring sequences of observable actions generated by the system. In this context, system behavior is represented by a set of sequences of actions observed during any program execution in the system. In our work we adapt the second approach and focus on monitoring system behavior represented in terms of sequences of system calls.

3.1. Overview

Our model for monitoring anomalous system behavior consists of a two-level classification mechanism (Fig. 1). Specification of normal and anomalous behavior represented in sequences or patterns of system calls are provided in the first level. In the event that a sequence to be monitored matches the specification, the second level classification is not invoked. The sequence that matches with legal specification is blocked and appropriate response actions are triggered. If the sequence is not found in the specification module, the second-level classifier is used. We then rely on a machine-learning technique to determine whether the sequence is normal or anomalous. In either case, the sequence is recorded in the corresponding specification (according to the machine learning classification result) for future reference. One of the important features of our model is that the technique can be deployed with empty or partial specifications in the first level. As more sequences are classified by the second level, the specifications are populated automatically. This flexibility reduces the overhead of developing the specifications manually.

Note that the specifications have to be developed and monitored during the detection phase separately for each program. Although the use of certain system calls might be similar across applications, the formed sequences might differ significantly.

In the following sections, we present the details of our approach.

3.2. Extended action graph: Exact

We model specifications in terms of Extended Action Graph (Exact) which is defined as follows:

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

A state in Exact corresponds to a system call. A sequence of system calls is therefore captured in Exact as $s_1, s_2, . . . , s_N$, where each $s_i$ represents a specific system call and has a transition to $s_{i+1}$ in Exact. Consider the example in Fig. 2. Each transition and the start states are labeled by a binary vector, e.g., $s_1 \rightarrow s_2$ and $L(s_1) = 101$.

It is worth mentioning here that not all the sequences in Exact are classified as valid. In general, the valid sequences form a superset of the known sequences (sequences from which the Exact was constructed in the first place). 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.

![Fig. 1 – Architectural model of our framework.](image-url)
A binary vector, whose $k$-th element is denoted by $s[k]$. If there exists a transition $s_i \rightarrow s_j$, where $s[k] = 1$, then $s_i$ and $s_j$ are said to be consecutive alphabets in the $k$-th known sequence. Note that the first sequence is identified by setting the first element to 1.

Validity takes care of (one or more) repetitions of the alphabet vectors. A sequence is valid if the binary vector of all transitions and the start state contain consecutive elements.

**Algorithm 1. (Monitoring procedure)**

1. function `seq monitor(seqIn, sk, E`)
2. if (seqIn == null)` then
3.  `seq = sk;`
4.  `start(seq) = sk;`
5.  `else`
6.  `seq = seqIn @ sk;`
7.  `start(seq) = start(seqIn);`
8.  `end if`
9.  `b0 = seqBad = false;`
10. `b0 = seqGood = false;`
11. `m = set(1);`
12. if (match(seq, m, Ebad)) then
13.  `seqBad = true;`
14.  `end if`
15. `m = set(1);`
16. if (match(seq, m, Egood)) then
17.  `seqGood = true;`
18.  `end if`
19. if (seqBad == true) then
20.  `if (seqGood == true) then`
21.    `delay = seq;`
22.    `return seq;`
23.  `else`
24.    `invokeResponse(seq);`
25.    `return seq;`
26.  `end if`
27. `else`
28.  `if (seqGood == true) then`
29.    `allow delay; delay = null;`
30.  `if (sk in start(SeqIn)) then`
31.    `start(sk) = sk;`
32.    `return sk;`
33.  `else`
34.    `allow sk;`
35.    `return seq;`
36.  `end if`
37.  `else`
38.    `delay = seq;`
39.    `invokeClassifier&updateExact(seq);`
40.    `return seq;`
41.  `end if`
42.  `end if`
43. `end function`

**3.3. Second-level classifier**

We rely on a machine-learning technique to classify behavioral patterns as normal and abnormal depending on how well they fit in the learned data domain. While any machine-learning technique that provides fast and accurate
classification can be applied as a second-level classifier, in our case study, we used support vector machines (SVM). SVM were initially introduced by Vapnik (1998) and have exhibited excellent accuracy on test sets in practice while having a strong theoretical motivation in statistical learning theory. There are two variations of the SVM algorithm: a supervised version, called two-class SVM and an unsupervised one referred to as one-class SVM. A supervised version of SVM works with labeled data sets and finds hyperplanes also called support vectors that maximally separate the data belonging to different classes (Vapnik, 1998). As opposed to two-class SVM, one-class SVM relies on separating all data from the origin using a hyperplane (Scholkopf et al., 1999). The decision to employ SVM for our experiments was primarily based on the availability of unsupervised option of the algorithm. Supervised algorithms generally require labeled data, i.e., data marked as normal or abnormal. As data labeling is time-consuming process and is dependent heavily on expert knowledge, reliance on the existence of labeled data may not be feasible for most production environments.

For the purpose of discussion, we illustrate the application of SVM classifier via an example. Let the observed input stream be $Istream = s_1, s_2, s_3, s_4, s_5, s_6, s_7$ and assume that Exact in Fig. 2 failed to recognize $Istream$ as a valid sequence.

First, we break-up $Istream$ following the transitivity relationship in Definition 2 of Section 3.2, i.e., $Seq_1 = s_1, s_2, s_3, s_4, s_5$ and $Seq_2 = s_2, s_3, s_4, s_5$. Note that the break-up point is at $s_2$ which appears in $Seq_1$ and $Seq_2$, and is the first alphabet repeated in $Istream$. This is done to appropriately insert the $Istream$ in Exact if both $Seq_1$ and $Seq_2$ are classified similarly (either both normal or both abnormal). 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 SVM is fed with subsequences: (i) $s_1, s_2, s_3$, (ii) $s_2, s_3, s_4$ (from $Seq_1$), (iii) $s_2, s_3, s_4$ and (iv) $s_3, s_4, s_5$ (from $Seq_2$). Finally, $Seq_1$ and $Seq_2$ are labeled as normal if and only if all their subsequences are classified by SVM as normal. Note that, break-up of $Seq_1$ and $Seq_2$ using sliding window does not adversely effect end result, i.e., if any subsequence of $Seq_1/Seq_2$ is classified as anomalous, then the corresponding sequence is conservatively classified as anomalous.

### 3.4. Monitoring for abnormal behavior

The monitor function in Algorithm 1 presents the pseudocode for our monitoring technique which uses the Exact algorithms and the machine-learning based classifier for identifying abnormal or anomalous sequence of system calls. It takes as argument, the sequence of system calls monitored and analyzed—$seqIn$, the new system call currently being monitored $sk$, and the Exact structures $E$ which is a tuple containing the Exact storing normal patterns—$E_{good}$ and the Exact storing abnormal or faulty patterns—$E_{bad}$. The function returns the $seq$, a sequence of system calls which will be used as the context in which subsequent system calls will be monitored.

If the $seqIn$ is null, than $sk$ is set as the first system call of the sequence $seq$ that will be monitored subsequently (Lines 1–4). Otherwise, $seq$ is synthesized by concatenating/appending $sk$ to $seqIn$ and the start state of $seq$ is set to start state of $seqIn$ (Lines 5–7). The sequence $seq$ is then checked to see whether it belongs to $E_{good}$ and/or $E_{bad}$ and the corresponding boolean variables $seqGood$ and $seqBad$ are appropriately updated (Lines 9–18).

Note that, a sequence may belong to both $E_{good}$ and $E_{bad}$ as there may be a prefix of a pattern that can be both normal and abnormal. In such a case (Lines 19–22), the delay variable is set to $seq$ indicating that (a) the sequence is not yet allowed to execute and (b) responses are not invoked for the sequence, until the subsequent calls allow to conclusively infer that it belongs to a normal pattern or an abnormal pattern. The function monitor returns $seq$ that will be used as the first argument for subsequent call to monitor with a newly arriving system call (the second argument).

In the event $seq$ belongs only to $E_{bad}$ (Lines 23–26), the response selection and deployment function invokeResponse is executed. In this case, monitor returns $seq$ so that subsequent call to monitor can use it to analyze new system calls in the context of $seq$ which, in turn, will allow invokeResponse to identify whether the response deployed on $seq$ had some impact or not.

Lines 27–37 correspond to the case when $seq$ is a normal pattern and as such, the sequence, stored in delay is allowed to execute (Line 29). Recall that, delay records the sequence that was not conclusively classified as normal or abnormal and is the suffix of $seqIn$. If $sk$ belongs to the start states of $seqIn$, then monitor returns a sequence containing $sk$ (Lines 30–32). This is because sequences in $Exact$ are constructed via transitive closure of substrings (see Definition 2). Otherwise, monitor allows the system call $sk$ to execute and returns $seq$ (Lines 33–35). The system call $sk$ is allowed in this case as it cannot be the start of an abnormal pattern of system calls (it is part of a normal pattern $seq$).

---

1 Response selection mechanism is beyond the scope of this paper. Please refer to (Stakhanova et al., 2007) for a taxonomy of responses and cost-sensitive response selection techniques.
Finally (Lines 37–40), if the $E_{\text{good}}$ and the $E_{\text{bad}}$ do not contain $\text{seq}$, then the $\text{delay}$ variable is set to $\text{seq}$ as the sequence cannot be classified by the $\text{Exact}$. The machine-learning based classifier is invoked and the classification result for $\text{seq}$ is used to update the $E_{\text{good}}$ or the $E_{\text{bad}}$.

4. Implementation overview

To evaluate the advantages of proposed approach we implement the $\text{Exact}$ structure and the corresponding algorithms. The architecture of the system implementation is presented in Fig. 3 and consists of the two components: $\text{Hybrid interception}$ and $\text{Detection module}$.

$\text{HISC}$, developed by Uppuluri et al. (2006), is a system call interception mechanism built for Linux architecture. In our framework we use HISC version for Linux kernel 2.4–18 (Uppuluri et al., 2006).

Detection module directly interacts with the HISC infrastructure and is responsible for monitoring and processing the system calls intercepted by the kernel-level components of HISC. The observed behavior in terms of sequence of intercepted system calls is matched with the normal and abnormal patterns/profiles contained in the $\text{Exact}$. In case an unknown sequence is encountered, the detection module processes the pattern through the classifier (i.e., SVM classifier) and makes necessary updates in the corresponding $\text{Exact}$. If the observed pattern represents abnormal behavior, detection module raises alarm.

The anomaly-based classifier in the detection module is implemented using libsvm tool, version 2.84 (Chang and Lin, 2001). The experiments are conducted on VMware Server 1.0.3 running Red Hat Unix 7.3 (Valhalla) Linux distribution with kernel 2.4.18-3 on IBM Lenovo T61 (Intel Core 2 Duo CPU T7300@2 GHz, 984 RAM).

5. Experimental results

We evaluate our framework through the set of experiments in the simulation environment and system prototype. The details of these experiments are presented in the following sections.

5.1. Simulation results

In this section, we discuss the experimental evaluation of our technique via simulations using publicly available synthetic sendmail data provided by the University of New Mexico (Forrest, 2005). Sendmail data consists of unlabeled collection of system calls. It includes a normal data sets consisting of only legal patterns and trace data sets containing normal patterns as well as anomalies. We consider three intrusion trace data sets most commonly analyzed by researchers: $\text{sendmailcp}$, $\text{decode}$ and $\text{fwdloops}$. The one-class SVM classifier is trained on the normal data set (training set), tested on the trace sets (test sets) with the window size of 8.

The question of selecting the right window size does not have any optimal answer and has been debated in the literature since the early use of fixed-length patterns. While working with the UNM data sets, the researchers refer to the work by Tan and Maxion (2002) that show that at least window size of 6 is necessary to capture all anomalies in UNM data sets. However, for other data most researchers tend to choose the size that gives the best performance. Following this practice, for the experiments with UNM data, we use various window sizes in the range from 6 to 12. While the best results are obtained with the window size of 8, the performance does not show any significant improvement or deterioration for other window sizes. Similarly, we select the window size of 3 for our implementation experiments given in Section 5.2.

5.1.1. Data sets

Table 1 presents the pattern of data being used for evaluation purpose in terms of number of sequences. The training data set contains 30792 normal fixed-length sequences. On the other hand, using $\text{Exact}$, the number of variable-length sequences is 3314. The decrease in the number of sequences is due to the fact that $\text{Exact}$ partitions sequences using repetitions and as such can handle variable-length sequences (see Section 3.2). At the same time, the number of unique variable-length sequences stored in $\text{Exact}$ is significantly smaller.

Table 2 presents the maximum length of binary vectors after building $\text{Exact}$ graphs on data sets which corresponds to the number of distinct variable-length sequences in $\text{Exact}$.

We then process 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 snsnmailcp, 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.

5.1.2. Efficiency

In these experiments we focus primarily on the rate of populating the Exact with normal (legal) and abnormal (anomalous) patterns. To evaluate our technique we monitor the stage at which each sequence is classified. We examine two scenarios:

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 are presented in Figs. 4–6. Fig. 4 shows the frequency at which both levels of classifiers (Exact and back-end SVM) are invoked for classifying the incoming sequences. Since simulation starts 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 are stored in the Exact. Consequently, the number of sequences classified at the Exact graph level increases. Fig. 5 shows the number of new patterns added to the empty Exact over the same run of decode trace set. The majority of patterns are recorded within about 200 sequences (out of 405 total sequences). After that, almost all patterns are found at the Exact level.

The result corresponding to the second scenario where the normal Exact graph is partially populated is shown in Fig. 6. As opposed to Fig. 4, the access rate of the second-level classifier at 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 are added to the specifications, the second-level classifier is still accessed whenever new normal or anomalous sequence is found.

In this scenario, we benefit from the available specifications having populated the Exact in advance. This shortens the start-up time necessary to store a sufficient number of patterns (Table 3). In fact, the processing time for 405 sequences is 2 times faster with the populated specifications (7 s) than with the empty specifications (16 s). It is worth mentioning that the SVM classifier access requires most of this time.

5.1.3. Accuracy

As the Exact graph provides a succinct representation of variable-length sequences learned via machine-learning technique, we focus in these experiments on the comparison of the accuracy of our structure to the accuracy of stand-alone SVM tested on a model built using the sliding window technique.

For evaluation purpose we consider detection rate (ratio of detected anomalies to the total number of anomalies...
The classification results are also given in terms of variable-length patterns stored by Exact (Table 5). Examining Table 5, we notice that prediction results are slightly different from the corresponding percentages given in Table 4. For example, the detection rate of Exact for snsnmailcp 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 effectively combined into one Exact sequence (according to procedure described in Section 3.3) resulting in an anomalous Exact sequence and thus providing a higher detection rate.

An opposite scenario is represented by decode intrusion, where the detection rate in fixed-length patterns is 100% which corresponds to 90 out of 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 back-end 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 consecutive Exact sequence to form a pattern of size 8 and performs classification. The combined sequence is classified by back-end SVM as normal resulting in 100% detection rate of the framework since this sequence is in fact present among normal fixed-length patterns. However, the Exact following SVM classification also records the combined sequence as two normal variable-length patterns unable to distinguish the subsequence of length 2 as abnormal. 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 percentages 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. At the same time, manual inspection of these results show that a number of FP sequences in Exact graph fully comes from the back-end 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 normal patterns reduces the number of false positives significantly (Table 5). Since the overall variability of sendmail behavior is small, even 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 number of detected sequences. This is because abnormal, incoming sequences are still recognized as unknown and processed by SVM algorithm as they would be if the Exact graphs were empty.

### 5.2. Implementation results

In this section, we focus on evaluation of our framework in the real system environment on the example of four Unix utilities. We experiment with the following commonly used Unix utilities (*Linux man page*, 2007): `cat` (used to concatenate and print files), `ls` (used to list files and directories), `mount` (used to

---

### Table 3 – Mean running time (decode intrusion. Average over 10 runs).

<table>
<thead>
<tr>
<th></th>
<th>Total time (s)</th>
<th>Back-end SVM running time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact with empty specs</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Exact with partially populated specs</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

---

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

<table>
<thead>
<tr>
<th>Stand-alone SVM</th>
<th>Our framework (results from the back-end SVM based on fixed-length sequences)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>snsndmailcp %</td>
</tr>
<tr>
<td>Detection rate</td>
<td>98</td>
</tr>
<tr>
<td>FP rate</td>
<td>11</td>
</tr>
</tbody>
</table>
request a file system on a separated device) and whoami (used to print an effective user name). The profiles for these programs are generated from test runs with different option configurations. In the implementation of our model we choose a window size 3 for two-class SVM.

5.2.1. Building normal behavioral profile

In this subsection, we explore the space and time usage of the detection engine. Space usage is primarily analyzed in terms of the size of the Exact required to capture normal behavior and the time usage is examined through the overhead in monitoring.

Figs. 7(a, b) and 8(a, b) show the population of the Exact with normal patterns for cat and ls utilities. Each of the Unix utilities (cat and ls) is executed with various options and the vertical partitions in the graphs show temporal ordering of their execution. For example, Fig. 7(b) shows that cat test is followed by cat -A test which is followed by cat -version, etc. The options that are not listed in the graphs (for either utilities) do not have any effect on Exact, i.e., these options do not provide additional behavioral patterns, and as such they are omitted from the figures.

The experiments reveal that behavior of ls has much more variation than that of cat. As a result, the number of states and the length of binary id (number of distinct Exact sequences) required to capture all possible ls behavior is larger than that required to capture the behavior of cat (21 states vs. 11 states and 56 Exact sequences vs. 26 Exact sequences respectively). The summary of the results for two other Unix utilities: mount and whoami, is provided in Table 6.

The monitoring overhead of the framework is examined on the example of four Unix utilities. Tests are run for 20 iterations. Each utility command is run without options. Populated Exact refers to the Exact containing normal patterns of Unix utilities executed without options. Back-end SVM denotes an anomaly-based algorithm triggered by our system for classifying unknown patterns.

Table 7 shows the execution time of our system with and without the existing normal profile in Exact. It is clear that the lack of any knowledge on the normal behavior of the program impacts the performance of our system. Specifically, the absence of the normal behavior in Exact on average doubles the running time. This increase in the system run time with empty Exact is primarily caused by the classification of the unknown patterns using the machine-learning algorithm.

For the utilities we considered, the deployment of our intrusion detection mechanism leads to average overhead of 16 ms. The overhead computation is equal to the difference between running time of the utility being monitored using our IDS with populated Exact (populated with normal behavior) and the running time of the utility alone. The overhead is caused due to two primary reasons. Firstly, the auxiliary module HISC, used to intercept systems, operates in single-threaded mode which allows monitoring system call sequence of one process at a time. Secondly, our detection mechanism is implemented in the user space. These two implementation decisions are made to allow quick development of the prototype of our framework. Currently, we are planning to address these drawbacks and deploy the IDS in the kernel space.

We have discussed in Section 2 a number of techniques that aim to develop specifications of programs automatically. As our method for generating specification and the resulting description of the specification are different from all the existing techniques, it is not possible to provide a direct

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

<table>
<thead>
<tr>
<th></th>
<th>Empty</th>
<th>Exact</th>
<th>Empty</th>
<th>Exact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ssn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>snsdmailcp</td>
<td>cp</td>
<td>decode</td>
<td>f</td>
<td>fdaloop</td>
</tr>
<tr>
<td>Number of detected seq.</td>
<td>24 of 24</td>
<td>90 out of 92</td>
<td>42 out of 43</td>
<td>24</td>
</tr>
<tr>
<td>FP sequences</td>
<td>21 out of 54</td>
<td>62 out of 313</td>
<td>75 out of 161</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 7 – (a) Exact states vs. legal patterns of cat utility. (b) Length of binary ID vs. legal patterns of cat utility.
comparison between our result and the result obtained using existing techniques. However, the work by Ko (2000) has similar objective as ours in terms of generating specification of programs based on the system calls executed by them. In his work, Ko presented partial specifications generated for several programs in terms of their specification lengths. Specification length of program refers to a set of various operations corresponding to system calls and their attributes where operations may include repetitive system calls if corresponding attribute information varies. For example as per Ko (2000), specification length of programs such as passwd is 14, at is 9 and atq is 2. In essence, the specification length captures the number of different types of parameter values associated with system calls executed by a program. Recall that, we do not capture such information; however, in contrast to Ko’s method which does not consider the sequence in which system calls are being invoked, our specification includes such information. In this sense, Ko’s method and ours can be viewed as complimentary approaches for developing specification. One of the future goals of our work will be to augment our specification with system call parameter values by integrating our method with the method proposed in Ko (2000). We have experimented with the same programs and generated Exact specifications. Our Exact specification for passwd program has 5 states and 7 unique Exact sequences (length of binary ID); while each of the specifications for at and atq programs has 3 states and 3 unique Exact sequences.

5.2.2. Detecting abnormal behavior
We investigate the capability of our detection engine to correctly identify abnormal behavior of program when its normal behavior is already known and stored in Exact. In essence, we are evaluating precision of Exact in capturing the known patterns with as little loss of information as possible such that abnormal behavior is not falsely classified as (i.e., matches with) patterns in Exact for normal behavior. In this context, we conduct two types of experiments described as follows.

5.2.2.1. Detection of new program behavior cat vs. mount. In this experimental setup, we populate Exact with normal behavioral patterns of cat utility as classified by SVM and then monitored the behavior of mount; the objective is to verify whether the behavior of mount utility can be captured as anomaly with respect to normal behavior of cat.

We select cat and mount mainly due to the nature of system calls generated by these two utilities. Fig. 9 shows the system calls made by the cat and mount utilities if executed in parallel. The difference in the system calls appears approximately after 200 system calls have been issued, i.e., initially the two utilities exhibit identical or close to identical behavior. As such variable-length sequences produced by Exact from the initial segment are also similar. However, once the behavior of these utilities becomes different, the detection component with Exact containing only normal profile for cat utility recognizes the execution sequences of mount utility as abnormal. This phenomenon is illustrated in Fig. 10(a, b) which demonstrates the growth of the Exact size in terms of number of states and the growth of binary id length (i.e., the number of distinct variable-length sequences) in the sequential execution of these utilities. After Exact is populated with patterns from normal behavior of cat utility, mount is executed. Since the initial behavior of these utilities is similar, the variable-length sequence, extracted from the monitored behavior of mount, fails to match sequences present in the Exact approximately after 400 system calls (which corresponds to 200 calls when utilities are executed in parallel, see Fig. 9). As a result, the backend SVM is invoked; it classified the new pattern as anomaly which is eventually stored in the Exact of abnormal patterns (represented by the increase of number of states and length of binary id in abnormal Exact).

5.2.2.2. Detection of abnormal behavior. As the examples of modified normal behavior of the program we experiment with two programs ls and whoami.

The motivation behind such experiments is to check whether our detection framework can deal with the trojan horse attack scenario. In this case, an intelligent attacker gains access to the system and modifies or leaves a modified version of the program (trojan) with some hidden (often malicious) features allowing an intruder to control the system after attacker leaves. As trojan horse programs are usually masked
under ordinary programs such as ls, find, etc (Dittrich, 2002), we experiment with detection of the novel patterns in the behavior of ls and whoami.

As an example, the following fragment has been added to the ls main() body of the program:

```c
# define MODE
(S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)
unsigned int nread, src, dst;
char *mybuff[100];
src = open("/etc/shadow", O_RDONLY);
dst = creat("/home/joe/myfile", MODE);
while((nread = read(src, mybuff, 100)) < 0){
    write(dst, mybuff, nread);
}
```

The code creates a copy of /etc/shadow file in the user’s directory with read and write permissions for the file owner and read permission for group and others. This code produces the following abnormal sequence of system calls:

```c
mmap2, open, open, read, write, read, write, ...
read, write, ioctl1, ioctl1
```

This modified version is tested against the profile gathered with the original version of the program given in Table 6. Our framework detects this sequence as anomalous when the system call write was produced. The first calls mmap2, open, open, read although represent the beginning of the abnormal pattern also present as a pattern in the normal profile.

Therefore, the detection engine delays the alert until the monitored sequence cannot be found in the normal profile and matched an abnormal pattern.

As another example of the new pattern detection in the modified programs we experiment with whoami utility. The following fragment has been added to the whoami main() body of the program:

```c
if (argc > 1 && (strcmp(argv[1], "--backdoor") == 0)){
    char *line[2];
    args[0] = "sh";
    args[1] = NULL;
    setuid(0); setgid(0);
    execve("/bin/sh", line);
}
```

The code allows user to type whoami-backdoor in command prompt to gain access to the shell prompt without supplying a password. This shell-code, i.e., code that gives shell to an attacker, is a common code used for the majority of local buffer overflows. It grants the access to the shell/bin/sh by calling execve() (Younan, 2003). This shell-code produces one abnormal sequence of system calls:

```c
write, setuid, setgid, execve
```

The first system call of this sequence write is not present in the normal profile of the original version of the program. As the result, the whole pattern is classified as anomalous. Our framework is able to detect it with the first anomalous call write.

5.2.3. Detection limitations

The proposed framework is pattern-based system, i.e., it detects the known patterns of normal or abnormal behavior represented by sequences of system calls. Therefore, if the anomalous behavior does not manifest itself through the change in the program’s system call behavior, it will not be detected. Naturally, some types of intrusions known for their ability to disguise the behavior as normal remain hidden from our system unless their intrusive behavior will cause program to behave differently, e.g., mimicry attacks, buffer overflow attacks.

We explore the behavior of our system for this type of attacks on the example of integer overflow vulnerability present in version 4.1 of ls utility.

This vulnerability can be exploited by providing ls with excessively large value for width -w option which causes an incorrect handling of internal integer variable (CVE-2003-0853, 2003).

The exploitable condition can be reproduced by typing the following line in the command prompt:

```
/bin/ls -w1073741828
```

This command requests a listing of the current directory sorted by columns with column position width 1073741828. The value provided for width variable is used in further calculations of the actual columns’ width. However, as the code does not provide necessary bound-checking of the accepted integer value, the program fails during the calculations causing a memory exhaustion.

The exploit produces the following behavior:
Since the system terminates the \texttt{ls} program, this vulnerability is non exploitable directly. However, due to the memory consumption it may cause remote denial of service through \texttt{wu-ftpd} program (CVE-2003-0853, 2003).

We execute this exploitable condition against the profile of \texttt{ls} normal behavior. Fig. 11 presents a stream of systems calls generated by \texttt{ls} utility during its normal execution and under integer overflow attack. System calls produced in each execution are mainly identical. The noticeable deviation from normal behavior during the exploit execution creates a long sequence of \texttt{brk} system call.

Although this exploit produces anomalous behavior, it is able to evade the detection engine due to two reasons: absence of system call arguments and lack of changes on the system call level. Our detection framework is not monitoring the input parameters of system calls mainly due to the overhead the analysis of the system call arguments often brings to the system (Giffin et al., 2004). However, to resolve this problem the proposed framework can be equipped with the dynamic or static analysis based checks of the input parameters (Wagner and Dean, 2001; Mutz et al., 2006).

Another reason for inability of our detection engine to recognize this anomalous behavior is the absence of any changes on the system call level. The abnormally long sequence of \texttt{brk} system calls after it is broken down into variable-length \texttt{Exact} sequences, constitutes a set of short legitimate patterns contained in normal \texttt{Exact}, which essentially allows the exploit to by-pass the detection engine unnoticed. This weakness of the framework is inherently present in all pattern-based approaches. To mitigate this limitation, in the recent years, several techniques have been introduced that allow to capture the intrusive behavior imitating the legitimate activity (Mutz et al., 2006). Our detection framework can be effectively enhanced with this type of techniques.

![System calls generated by cat & mount utilities](image)

Fig. 9 – System calls generated by cat and mount utilities.

![Growth of Exact graph](image)

![Growth of binary ID length](image)

Fig. 10 – (a) \texttt{Exact} states vs. legal patterns of \texttt{ls} utility. (b) Length of binary ID vs. legal patterns of \texttt{ls} utility.
6. Conclusion

In this work we present a technique which effectively combines the advantages of anomaly-based and specification-based approaches recognizing a known behavior through the specifications of normal and abnormal patterns and classifying unknown patterns using a machine-learning algorithm. Such combination allows adaptation of the specification-based detection to the new patterns, and provides an effective method for automatic development of specifications. We develop a novel data structure, Extended Action Graph (Exact), for effectively storing variable-length patterns as specifications and present the associated algorithms for insertion and detection of patterns in the structure.

In addition to simulation experiments, we provide a prototype system implementation. The experimentation with the system prototype show us the advantages of using a combination of specification-based and anomaly-based techniques for detection of abnormal behavior. Since the monitoring for anomalous patterns in programs' behavior is based on the profile of normal behavior, the effectiveness of the detection can be characterized by the following factors:

- The specifications of normal behavior: Our system is able to generate specifications of normal behavior automatically. In our system prototype, we develop normal profiles for four Unix utilities in a short period of time. Although considered programs have very simple behavior, we believe that the profiles for more complex software applications can be generated automatically in a similar fashion.

- Adaptation to new behavior: Our system effectively learns new program behavior on the example of Unix utilities and is able to recognize previously unknown behavior as normal or abnormal.

- Detection of unknown anomalous behavior: Our system correctly recognize the behavior of mount utility as abnormal based on the profile of normal behavior of cat utility. Since the system does not have any information on the mount utility, its behavior is unknown to the system and is classified as abnormal with respect to known normal behavior of cat utility.

- Detection of known anomalous behavior: We perform several experiments on the detection of programs anomalous behavior of whoami and ls utilities based on their normal and abnormal profiles. Our system successfully detects the attacks which caused changes in the normal system-call flow of the programs.

6.1. Future work

Our research opens various opportunities for future research avenues. One direction we plan to explore is the deployment of countermeasures as a result of anomalous behavior detection. The proposed Exact structure allows easy integration of intrusion detection process with the response mechanism. Due to the nature of the proposed detection technique, the response component will have the ability to deploy the response actions proactively before the detected intrusion completes. We also plan to further enhance the efficiency of the proposed approach by combining the Exact structures for normal and faulty specifications. Such a combination may support early identification of sequences as normal or abnormal. This may be especially applicable in the domain of adaptable software systems. The main aim of adaptability is to identify abnormalities and apply appropriate adaptation to avoid failures. Our technique would allow abnormal specifications to be annotated with corresponding adaptations paving the way to efficiently identify and apply
adaptation automatically for similar/identical patterns of abnormal behavior.

REFERENCES


Dittrich D. “Root Kits” and hiding files/directories/processes after a break-in. Available from: http://staff.washington.edu/dittrich/misc/faqs/rootkits.faqs; May 2002 [accessed 15.05.07].


Linux man page. Available from: http://linux.die.net/man; 2007 [accessed 15.05.07].


Tseng, C.-Y., Balasubramanym, P., Ko, C., Limprasittiporn, R., Rowe, J., Levitt, K., A specification-based intrusion detection


Natalia Stakhanova is a Research Scientist at Information Security Centre of eXcellence (ISCX) Faculty of Computer Science, University of New Brunswick. She received her Ph.D. degree in Computer Science from Iowa State University, USA. She has extensive research experience in intrusion detection and general information security field. Natalia has published numerous journal and conference papers and has been the recipient of the “Nokia Best Student Paper Award” at The IEEE International Conference on Advanced Information Networking and Applications (AINA) in 2007. She has two pending patents in the area of network security.

Samik Basu is an Associate Professor in the Department of Computer Science at Iowa State University, Ames, Iowa. He received his Ph.D. in Computer Science from the State University of New York at Stony Brook in 2003. His primary research area is formal verification of systems exhibiting infinite-state behavior. He also actively participates in projects that involve application of formal (logic and/or automata-based) methods in intrusion detection, web service composition and diagnosis/synthesis of controllers for discrete event systems.

Johnny Wong is a Professor & Associate Chair of the Computer Science Department at Iowa State University. His research interests include Software Systems & Networking, Security & Multimedia Systems. Most of his research projects are funded by government agencies and industries. He is the President/CEO of a startup company EndoMetric, with products for Medical Informatics. He has served as a member of program committee of various international conferences on intelligent systems and computer networking. He was the program co-Chair of the COMPSAC 2006 and General co-Chair of the COMPSAC 2008 conference. He has published over 100 papers in peer reviewed journals and conferences. He is a member of IEEE Computer Society and ACM.