Available online at www.sciencedirect.com
ScienceDirect
Pattern Recognition 41 (2008) 90–101
PATTERN RECOGNITION
THE JOURNAL OF THE PATTERN RECOGNITION SOCIETY
www.elsevier.com/locate/pr

Probabilistic suffix models for API sequence analysis of Windows XP applications

Geoffrey Mazeroff[a,*], Jens Gregor[a], Michael Thomason[a], Richard Ford[b]

[a]*Department of Computer Science, University of Tennessee Knoxville, 203 Claxton Complex, Knoxville, TN 37996-3450, USA*
[b]*Department of Computer Sciences, Florida Institute of Technology, 150 W. University Blvd., Melbourne, FL 32901, USA*

Received 20 March 2006; received in revised form 9 March 2007; accepted 13 April 2007

Abstract

Given the pervasive nature of malicious mobile code (viruses, worms, etc.), developing statistical/structural models of code execution is of considerable importance. We investigate using probabilistic suffix trees (PSTs) and associated suffix automata (PSAs) to build models of *benign* application behavior with the goal of subsequently being able to detect malicious applications as anything that deviates therefrom. We describe these probabilistic suffix models and present new generic analysis and manipulation algorithms. The models and the algorithms are then used in the context of API (i.e., system call) sequences realized by Windows XP applications. The analysis algorithms, when applied to traces (i.e., sequences of API calls) of benign and malicious applications, aid in choosing an appropriate modeling strategy in terms of distance metrics and consequently provide classification measures in terms of sequence-to-model matching. We give experimental results based on classification of unobserved traces of benign and malicious applications against a suffix model trained solely from traces generated by a small set of benign applications.

Keywords: Probabilistic suffix model; API sequence classification; Anomaly detection; Agglomerative clustering; Windows XP; Malicious mobile code; Virus; Worm

1. Introduction

In today's society the transfer of digital information is commonplace and often crucial for the domain at hand. Within this information can be embedded executable code whose intent or purpose is hidden from the recipient. One example is *malicious mobile code* that can compromise a user's machine and/or data and also propagate to other machines [1]. Several types of malicious mobile code have emerged since the 1980s [2]; consequently, it is quite common to have experienced (or know someone who has experienced) its negative effects. One notable instance of malicious mobile code, the Slammer worm, was able to spread globally in a matter of minutes and had drastic real-world consequences [3]. In the 2005 CSI/FBI Computer Crime and Security Survey, viruses have been reported to be the principal cause of financial loss to companies affected by computer crime [4].

In addition to being pervasive, malicious mobile codes tend to change in how they behave over time. A detection system must therefore be able to handle known as well as *unknown* threats. We present a method of "fingerprinting" executable code by analyzing sequences of system calls generated by an application, i.e., inferring its typical behavior. The goal is to ultimately be able to detect that an application starts behaving differently than previously observed. In other words, we aim to distinguish between normal and abnormal behavior. Although not explored in this paper, this approach is particularly applicable to spyware. For example, if the behavior of one's web browser changes significantly, this may indicate the presence of spyware.

Modeling sequences of system calls for the purpose of detecting anomalous activity is not novel. Forrest et al. have studied a machine-learning approach using Unix system call sequences

* Corresponding author.
E-mail addresses: mazeroff@cs.utk.edu (G. Mazeroff), jgregor@cs.utk.edu (J. Gregor), thomason@cs.utk.edu (M. Thomason), rford@se.fit.edu (R. Ford).

0031-3203/$30.00
doi:10.1016/j.patcog.2007.04.006

[5,6]. Bayesian networks have been used on Unix-based data by Schonlau [7], Schultz et al. [8], and Feng et al. [9]. Similarly, Giacinto et al. [10] have studied a combination of classifiers (including Bayes) to detect intrusion and DuMouchel et al. [11] have investigated the use of principal component analysis. Neural networks have also been employed to classify abnormal behavior as shown by Ryan et al. [12] and Tesauro et al. [13]. Using Markovian models (including hidden Markov models) in this problem domain has been explored by Ju and Yeung et al. [14,15]. Recently, Heller et al. [16] investigated support vector machines applied to Windows-based data. A comparison with any of the above is beyond the scope of the present paper.

Even though research on this topic is extensive, the problem of detecting malicious mobile code is still difficult. Developing statistical/structural models of sequential system calls is of considerable interest in practice. A model of benign code behavior gives insight into normal usage, common-cause events, and similar characteristics. In addition, statistically significant deviations from the model may signify anomalous, and potentially malicious, code behavior. In these cases the user can be prompted to consider counter-measures such as terminating execution of the offending program, and undoing changes to the file system or other system resources such as the registry database on a Windows system.

For Windows applications, Whittaker and de Vivanco [17] proposed (i) intercepting the low-level system calls made to the operating system, e.g., when a program attempts to read or write a file, access or modify a registry key, allocate and deallocate memory, dynamically link with a runtime library, create a child process, and so forth; and (ii) comparing these actions against a model of *benign* behavior. The associated software package, which allows a Windows application to be monitored in real-time, has become known as Gatekeeper [18]. Gatekeeper records the Windows API (application programmer interface) calls that the monitored application makes. These system calls are used by Windows applications to carry out specific tasks (e.g., `CreateProcess` invokes a new process and commences its execution, `DeleteFile` removes a specific file from the file system, and `RegOpenKey` opens a Windows registry key for use). The information associated with each API call within Gatekeeper is a process ID, a timestamp, a function name, the function parameters, and the return value. We obtained the data analyzed in this paper by using Gatekeeper to monitor the Windows API calls generated by an application. This information is processed first for model inference, then for classification of the application behavior as either benign or potentially malicious.

In a previous paper [19], we presented preliminary work on how the trace information produced by Gatekeeper can be symbolically encoded in a manner that facilitates model building in the form of the probabilistic suffix trees (PSTs) and automata (PSA) proposed by Ron et al. [20]. (Other recent applications of probabilistic suffix models include protein families [21] and musical styles for machine improvisation [22].) In this paper we present a more sophisticated scheme for carrying out the symbol encoding of code traces together with a number of new generic techniques for analyzing and manipulating the suffix models.

Following a concise description of the PST/PSA inference algorithms, we present an algorithm for retroactively reconstructing a full PST from the pruned version normally stored. We then give an algorithm for merging two PSTs to produce the PST that would have been inferred had all the learning samples been processed simultaneously. One advantage thereof is that new learning samples can be incorporated on an incremental basis as they become available rather than having to re-infer a new model from scratch each time. Perhaps more importantly, access to the full PST also allows the distance between PSTs to be defined which in turn supports model comparisons, analysis, and clustering. Finally, an algorithm for matching sequences against an inferred PSA is described. This algorithm has application to determining the probability of a given sequence as it corresponds to a given model. We close by presenting experimental results that illustrate (a) how suffix models trained using sequences from benign and malicious applications are related with regard to distance-based clustering, and (b) how unobserved sequences from benign and malicious applications similarly relate to a suffix model trained solely on benign code traces.

2. Probabilistic suffix models

Two probabilistic suffix models are employed for analyzing API traces: the PST and the PSA. To make the paper self-contained and facilitate the description of the model manipulation, analysis, and use in the next section, we give a concise description of both models including general statements about algorithmic time-complexities. The reader is referred to the seminal paper by Ron et al. [20] for step-by-step algorithmic details and a formal time complexity analysis.

2.1. Probabilistic suffix trees

A PST is an n-ary tree whose nodes are organized such that the root node gives the (unconditional) probability of each symbol of the alphabet while nodes at subsequent levels give next-symbol probabilities conditioned on a combination of one or more symbols having been seen first. The *order* of the model is analogous to the depth of the tree: A tree of order L has L levels beyond the root. Furthermore, the nodes on the bottom-most level retain a "history" of L symbols. The next-symbol probabilities are computed as relative frequency-count estimates of symbol occurrences in the learning sample(s).

To illustrate, Fig. 1 shows the second-order PST inferred from the sample "$ `1 2 3 1 2 3 2 1 3` $". (The $ symbol delimits the start and end of sequences and is used solely for bookkeeping in the algorithms described later in this paper.) Beginning with the root node which represents the empty string, each node is a *suffix* of all its children—hence, the name of the model. The numbers in parentheses adjacent to the nodes are the conditional next-symbol probabilities. Note also that next-symbol "transitions" jump from one branch to another, not from

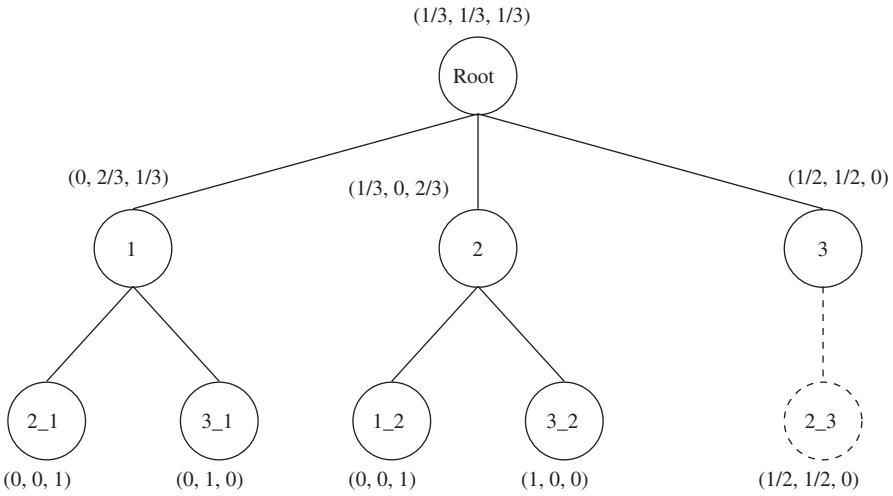

Fig. 1. Example of an inferred PST.

a parent node to its children. This transition pattern is due to the suffix format of the node labels.

The recursive algorithm for inferring a PST is summarized as follows. First, the sample(s) are scanned to determine the probability of each symbol. Next, each symbol with non-zero probability becomes a child node of the root, and for each such node the sample is rescanned to determine the next-symbol probability distribution. This step may create new leaf-nodes and add another level to the PST thereby increasing the model order. The inference recursively adds additional levels by checking each current leaf-node to determine whether new leaf-nodes must be created as its offspring. During the rescans, information is stored corresponding to the symbols that precede the current leaf-node as these symbols may appear in labels of children thereof. A node is added only if the substring of symbols that corresponds to the node label is in the learning sample(s) and if a non-zero next-symbol probability distribution exists for it. Some branches typically die out early while other branches propagate to the maximum depth of the tree. Thus, a PSTs *variable memory length* [20] is determined by patterns in the learning data itself.

A recursive bottom-up pruning process may now be carried out to remove leaf-nodes that provide statistical information which is similar to that of the parent nodes. This process of eliminating "same-as-parent nodes" may potentially remove many nodes thus creating an attractively parsimonious model. The pruning is based on the probability information, not the frequency counts themselves. The pruned model is advantageous because it captures sequence information in a compact form, which increases algorithmic and space/storage efficiency. In our implementation, a leaf-node is pruned only if it carries the *exact* same next-symbol probability distribution as its parent node. As shown later, this restriction will allow us to reconstruct the PST as it appeared before any pruning took place.

The overall time complexity for inferring and pruning a suffix tree is $O(Ln^2)$ where n is the total length of the learning sample(s) [20]. We have opted for a more efficient implementation. Specifically, we set an upper bound on L, the model order, prior to inference. This allows us to build a balanced red–black search tree [23] that stores frequency counts for all observable symbol sequences in the input data. Creating the PST then becomes a simple matter of looking up each node label in the red–black tree. Because the time complexity of red–black tree insertion and lookup operations is logarithmic, the resulting PST inference algorithm has a time complexity of $O(Ln \log n)$. That said, the reader should note that this paper is not intended to focus on improving the computational efficiency for PST inference. Indeed, a sophisticated PST algorithm has been proposed that achieves $O(n)$ performance [24]. This algorithm and other potentially more efficient algorithms available in the literature are beyond the scope of the present paper.

2.2. Probabilistic suffix automata

A PSA can be inferred directly (i.e., without access to the inference sequences) from a PST. The recurrent states within the PSA form a discrete-parameter, recurrent Markov chain of variable-order for which descriptive statistics such as steady-state distributions and mean transitions can be computed. Another property of the PSA is that it can be used to generate sequences based on the learning data from which it was inferred.

Fig. 2 shows the PSA inferred from the PST in Fig. 1. The example PSA contains three transient states (i.e., "Root", "1", and "2") and five recurrent states ("1_2", "3_2", "3", "3_1", and "2_1"). The values on the arcs connecting two given states represent the probability of transitioning from one state to the next. Transitions may only occur where arcs are present.

Before describing the algorithm for inferring the suffix automaton, it is useful to note that the nodal relationship in the automaton differs slightly from the relationship in the suffix tree. Parent nodes are considered *predecessor states* and

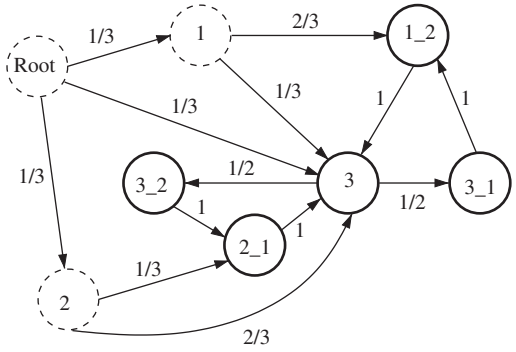

Fig. 2. PSA inferred from the PST in Fig. 1. States in dashed outline are transient. States in bold outline represent the recurrent states.

child nodes are considered *successor states*. The concept of a predecessor is based on excluding the right-most symbol. For example, the predecessor to the state "1_3_2" is "1_3", as opposed to the parent node of "1_3_2" being "3_2" in the suffix tree.

The probabilistic automaton is created entirely from the information provided in the suffix tree without any direct reference to the learning sequences. First, add all of the leaf-nodes in the suffix tree to the automaton as recurrent states. Second, create a state that corresponds to the root node of the suffix tree, and then "flood" downward from that state to all of the states created in Step 1. For example, to connect the root state with "1_3_2", two intermediate states ("1_3" and "3") are connected from the root state to "3", to "1_3", and finally to "1_3_2". Third, create any necessary arcs between states. This process is accomplished by visiting each state in the automaton and looking at its next-symbol probability (available in the PST). If a given symbol has a non-zero probability of occurrence after the given state label, append the symbol to the end of the state label and remove symbols from the *front* of the new label until a state is found that exists in the automaton. For example, state "3_2" in Fig. 2 has a non-zero probability of encountering a next symbol of "1"; therefore, a new label is created, "3_2_1" and symbols are removed from the left-side until a match is found. Removing the "3" yields "2_1" which is a defined state, so an arc is created between "3_2" and "2_1". Finally, assign state types to the states created in Steps 2 and 3. This is accomplished by visiting each of the created states and determining if any edges incident upon them originated from recurrent states. If this is the case, the given state is marked as recurrent. For example, state "3" is recurrent because there is an incident arc from "2_1" which was defined as a recurrent state in Step 1. Any states not marked recurrent after visiting each state are labeled as transient by default.

The time complexity associated with converting a PST into a PSA is $O(Ln^2)$, where L and n are defined as above [20]. This, however, is based on the worst-case scenario of a maximally connected automaton. In practice, we find that only a few states can be reached from any one state, which leads to $O(Ln)$ behavior in practice.

3. Model manipulation, analysis, and usage

3.1. Retroactive full PST reconstruction

There are two primary types of PSTs, namely, *full* and *pruned* which refer, respectively, to the trees that result from the inference process *before* and *after* "same-as-parent" pruning has taken place. By storing the next-symbol frequency information associated with each node in addition to the derived probability distributions normally considered, and by introducing unique start and end symbols used for book-keeping purposes, it is possible to retroactively reconstruct the full PST from its pruned counterpart without access to the learning data. This reconstruction occurs only up to the preset maximal order of the given PST.

Algorithm. 1. $T_F = \text{ReconstructPST}(T_P, L)$

$T_F[\text{root}] = T_P[\text{root}]$
for each level $\ell = 1 : L$ **do**
  for each possible node label $\sigma = \sigma_1 \cdots \sigma_\ell$ **do**
    if $T_P[\sigma]$ exists **then**
      $T_F[\sigma] = T_P[\sigma]$
    else
      $f = T_P[\sigma_1 \cdots \sigma_{\ell-1}].\text{freq}[\sigma_\ell]$
      if $f > 0$ **then**
        Create missing node $T_F[\sigma]$
        $T_F[\sigma].\text{prob}[\ ] = T_P[\sigma_2 \cdots \sigma_\ell].\text{prob}[\ ]$
        $T_F[\sigma].\text{freq}[] = f \times T_F[\sigma].\text{prob}[\ ]$
      end if
    end if
  end for
end for

Algorithm 1 provides pseudo-code for the level-by-level scan of the pruned PST that allows the reconstruction to take place. Each missing node label is tested to see whether it was excluded because the particular symbol combination did not exist in the learning data (in which case the given node is ignored) or because it was pruned because it had a same-as-parent next-symbol probability distribution (in which case the node is reinserted into the tree). The test is based on the next-symbol information of the corresponding prefix node. The new node's frequency counts are reconstructed by finding the number of times the current node's rightmost symbol is encountered based on its prefix, then multiplying that count by the probability distribution for the parent node. In other words, missing node X_Y_Z will be added if the node labeled X_Y has a non-zero next-symbol frequency count for symbol Z. Furthermore, we can obtain the entire next-symbol frequency count distribution for node X_Y_Z by multiplying the next-symbol probability distribution associated with node Y_Z by the aforementioned non-zero next-symbol frequency count for symbol Z.

As a concrete example, consider Fig. 1 where node "1_1" is missing. The next-symbol frequency information for the prefix "1" shows that the input data used for inference did not contain two consecutive "1"s. This node should consequently remain

absent from the full PST. Node "2_3" is also missing. The node corresponding to the prefix, namely "2", indicates that the symbol "2" can be followed by "3". This node should therefore exist in the full PST with the same next-symbol probability distribution as listed for node "3", its parent. Now, because symbol "3" follows symbol "2" twice (based on the frequency counts stored at node "2"), we multiply the next-symbol probability distribution by two to reconstruct the missing next-symbol frequency counts.

The above algorithm guarantees reconstruction of all missing nodes up to the level corresponding to the original maximum model order, but not beyond although in some cases more levels can be reconstructed. This guarantee holds because the PST can be thought of as a placeholder for frequency counts. The sequence delimiter symbol ($) ensures that the last non-$ symbol in the sequence is included and that the frequency counts of all non-$ symbols are identical to the learning sequence. Thus the inherent structure of the suffix tree allows next-symbol dependencies (up to the original inference level) to exist and therefore be used to reconstruct missing information.

As written, the time complexity for Algorithm 1 is $O(Ln)$ because that is the bound on the number of nodes in the full PST and each node is considered once. In reality, the inner for-loop needs only iterate over node labels for which the prefix corresponds to an actual node at the parent level. The stated time complexity is thus a worst-case bound.

3.2. PST merging

Frequency count statistics are extracted from the learning samples during inference and are incorporated into the model one sample at the time. As described above, pruned information can be recovered. This recovery allows an algorithm to be formulated for *merging* one PST with another simply by adding the next-symbol probability and frequency count distributions of the individual nodes in the equivalent full PSTs.

Algorithm. 2. $T_M = \text{MergePSTs}(T_{P1}, T_{P2}, L)$

$T_{F1} = \text{ReconstructPST}(T_{P1})$
$T_{F2} = \text{ReconstructPST}(T_{P2})$
for each node label σ in T_{F1} and/or T_{F2} **do**
  if $T_{F1}[\sigma]$ exists **then**
    $T_M[\sigma] = T_{F1}[\sigma]$
    if $T_{F2}[\sigma]$ exists **then**
      $T_M[\sigma].\text{freq}[] = T_M[\sigma].\text{freq}[] + T_{F2}[\sigma].\text{freq}[]$
      Recalculate next-symbol probabilities for $T_M[\sigma]$
    end if
  else
    $T_M[\sigma] = T_{F2}[\sigma]$
  end if
end for
$T_M = \text{PrunePST}(T_M)$

Algorithm 2 provides pseudo-code for the merging process. As we step through the nodes in the two trees, we add the next-symbol frequency count distributions for nodes that exist in both and copy in nodes that are unique to either tree. The next-symbol probabilities are recalculated as the ratio of each new next-symbol frequency count by the total frequency count now associated with the node.

One advantage of PST merging is that it allows for incremental learning. That is, when a new learning sample becomes available, we can build a PST for it and then carry out a merger with the existing PST. Not only can this be done without having access to the entire potentially large collection of learning samples, it can also be done faster than re-inferring a new PST from scratch for all these samples plus the new one. Suppose that the original learning data consist of a concatenated sequence of samples n symbols long and that the new sample is m symbols long. From above, we know that a new PST can be inferred from scratch in time $O(L(n+m)\log(n+m))$. In contrast, we can build a PST for the new sample in time $O(Lm\log m)$ and subsequently merge it with the existing PST in time $O(L(n+m))$. The latter cost comes from first reconstructing and then iterating over full versions of the two PSTs, both of which are linear-time operations. This indicates that incremental learning is the more efficient approach of the two when $m \ll n$, a condition that is easily met in practice.

3.3. PST comparison

For the purpose of computing the *distance between two PSTs*, it is convenient to consider vectorizing the nodes of a full PST. That is, any consistent, non-negative numbers assigned to the PST nodes—such as next-symbol frequency counts or conditional probabilities computed as the relative frequencies—can be viewed as the coordinates of specific axes in an underlying vector space. This allows the distance between two PSTs to be computed as any vector distance, e.g., L_1, L_2, or L_∞, in either the full space or in any subspace that makes sense to consider.

In this paper we propose a Euclidean distance based on a joint-probability distribution at the level of the leaf-nodes of two PSTs because this is closely related to computing the distance between the two sets of recurrent states that constitute the corresponding PSAs. An unconditional probability distribution over leaf-nodes is obtained by computing the match probability of each leaf-node's label, then normalizing these to sum to one. The label match probability is the product of the probabilities associated with making the necessary transitions to match the sequence of symbols that make up the label starting from the root node. Let T be a PST and let $p(i)$ and $p(k|i)$, respectively, be the match probability for leaf-node i and the conditional next-symbol probability for symbol k where $p(i) = 0.0$ if the node is non-existent. Then the following distance, which is Euclidean and thus a metric, is defined between T_1 and T_2, which are two full PSTs:

$$d(T_1, T_2) = \sqrt{0.5 \sum_i \sum_k (p_1(i)p_1(k|i) - p_2(i)p_2(k|i))^2}.$$

We note that the distance between two identical trees is 0.0, while the distance between two trees that share no common

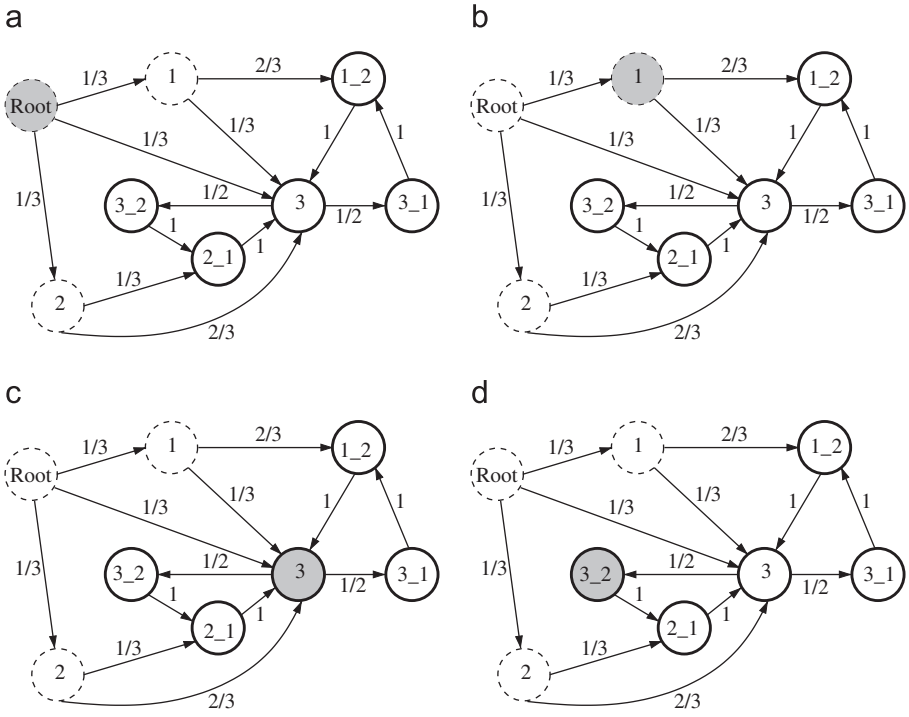

Fig. 3. Matching a sequence against a PSA.

nodes is 1.0. With respect to the notion of using $p(i)$, the idea is to eliminate some of the "noise" that low-probability nodes might otherwise induce. For example, two nodes that have vastly different next-symbol probability distributions should not have a great impact on the overall distance between two trees if these nodes have a low probability of being reached. In other words, we let nodes associated with high match probabilities carry more weight in the distance calculation.

A distance measure leads naturally to ideas of *clustering* among PSTs. A straightforward approach is to infer a PST for each sample, compute all pair-wise distances, and then apply a hierarchical agglomerative algorithm (see e.g. [25]) to determine the set of clusters. With hierarchical clustering, nested sets of clusters are created, with one top-level (root) cluster that includes all features (i.e. PSTs). Each level below the root cluster represents a more fine-grained set of clusters. At the lowest level, each feature is its own unique cluster. The agglomerative descriptor indicates that the clusters are formed in a bottom-up fashion: The PSTs (or clusters of PSTs) that have the smallest pairwise distance are grouped together.

We can combine this clustering approach with the concept of merging described above. The PSTs that have the smallest distance between them are merged. Next, the pairwise distances are recomputed such that the individual trees (just merged) are removed and the merged tree put in their place. This process continues until a desired distance threshold is reached or until a single tree is produced (as a result of merging all individual trees). This *dynamic* clustering approach allows us to combine PST analysis techniques to determine if and how clusters form within the represented datasets.

3.4. Probabilistic automaton matching

Matching samples against a probabilistic automaton yields the cumulative probability of the given sample being generated by the PSA. To match a sample against an automaton, start with the first symbol and find its corresponding state in the automaton. Continue by traversing the sample, appending the next symbol onto the previous one. Next, attempt to find the state based on the newly created label. If the state exists, note the probability of transitioning from the previous state to the current state and continue appending symbols from the sample. If the state does not exist, however, remove one symbol at a time from the front of the target state label until a state is encountered that does exist in the automaton. In the worst-case, removing prefix symbols will yield a label of a transient state because that state should contain the next-symbol probability to proceed with the matching algorithm. In other words, the next state used in the matching process could be the current state's successor or simply another state (derived by removing prefix symbols) that allows the matching process to continue.

To match the sample "1 3 2" against the PSA in Fig. 2, the following steps would occur (see Fig. 3). Step 1: state "1" exists, so transition from the root to "1" with probability 1/3. Step 2: append "3" to the current state to obtain "1_3", which does not exist. Begin removing prefix symbols until a valid state label is obtained. In this case, "3" exists, so transition from "1" to "3" with probability 1/3. Step 3: append "2" to the current state to obtain "3_2", which exists. Transition to "3_2" with a probability of 1/2. The cumulative probability of match is 0.0556.

The use of matching samples against suffix automata is analogous to that of matching samples against suffix trees. The higher probability of match, the more likely it is that the automaton would have generated the given sample. For example, the probability of match for the sample "1 3 3" is zero, which is less likely than the previous example "1 3 2". The zero probability of match is verified by noting that two adjacent 3's never occur in the learning sample. Various methods of dealing with such zero-probability scenarios have been studied but are not within the scope of this paper.

Matching a sample of length m against a PSA is an $O(m)$ operation and thus quite fast.

4. Applying suffix models to API sequences

We now apply the suffix models and analysis techniques to a suite of applications monitored by Gatekeeper. The following subsections provide a brief description of the data and how it is reduced, a discussion on model inference choice based on PST distances, a description of match probability smoothing, and finally classification results based on PSA matching.

4.1. API sequences

Our benign Windows XP application experimental dataset consists of 62 code traces from 37 different programs such as Microsoft Word, Excel, Internet Explorer, and Adobe Acrobat Reader. The runs range in complexity from opening and closing an application to performing normal tasks including opening files, manipulating documents, and changing various software settings. The malicious application dataset contains code traces of 150 different variants of malicious code (i.e., viruses and worms), a subset of those active in March 2003 [26]. Their behavior ranges from deleting files to manipulating the registry and using network services to gain access to other computers. The Wildlist samples are used because they are instances of threats that actually replicate. Several *zoo* samples (i.e., "viruses that only collectors or researchers have seen") exist from other sources [1]. However, these may be contrived or partial applications that are not representative of the actual threat. By using Wildlist samples, we are testing our methods against the threats that would most likely be encountered in the real world.

The benign application dataset contains code traces with lengths from 3K to 139K API calls. The malicious application dataset contains traces are significantly shorter with lengths from 100 to 1000 API calls. This discrepancy in length can be attributed to (i) the inherent behavior of malicious mobile code wherein the typical objective is to have the harmful events occur shortly after the application's execution, (ii) premature termination of the application, and (iii) the majority of the application's lifetime being spent simply waiting (i.e., sleeping) for a specific trigger event to occur.

The main focus in determining what portions of the traces are to be used is based on the set of API calls capable of changing the state of the registry or file system (i.e., creation, modification, and removal). Next, the parameters of those calls are analyzed to determine where specifically in the registry or file system changes are being made. The following is an example of a call to the API `RegOpenKeyEx` from a Gatekeeper log of Microsoft Excel:

```
412, 1565870067586, 19:27:16:5,
RegOpenKeyEx, 63, hKey,
2147483650, lpSubKey, Software\Microsoft
\Windows\CurrentVersion\
Internet Settings\ZoneMap\Domains
\microsoft.com, phkResult, 0,
retCode, 2, 0, 0
```

In this case the application is requesting that the following registry key be opened:

```
HKLM\Software\Microsoft\Windows
\CurrentVersion\Internet Settings\
ZoneMap\Domains\microsoft.com
```

As the population of such locations is immense and highly-variable, an effort is made to generalize the sample space. Dominant prefix strings are discovered such that only most significant levels of the registry and file system hierarchies are preserved. This reduction is accomplished by compiling all registry keys and file paths into two respective groups, thus representing the sample space of observed hierarchies. Next, each level of the hierarchies is examined to determine which children occur with probability $P(s) > 0.1$, where s is the text-string being considered.

For example, the registry key above reduces to

```
HKLM\Software\Microsoft\Windows
\CurrentVersion\Internet Settings
```

The original registry key had a probability of 0.0004. After removing `microsoft.com`, the probability increased to 0.0082. After removing `Domains`, the key occurred with a probability of 0.0291. Finally, removing `ZoneMap` gives a probability of 0.117.

Specific file names as well as registry values do not currently play a role, as the primary emphasis lies in inferring where an application operates, not what specific changes it makes to the system. One should also note that threshold value for $P(s)$ is highly data-dependent. If the threshold is too high, the resultant set will be very coarse. Likewise if the threshold is too low, there will be many unique text-strings that can potentially add noise to the alphabet. Our primary goal here is to prove the effectiveness of the technique with the caveat that an expansion of the training dataset would prove useful.

All application traces are as described above. The result is approximately 80 unique strings which are mapped into distinct integer values.

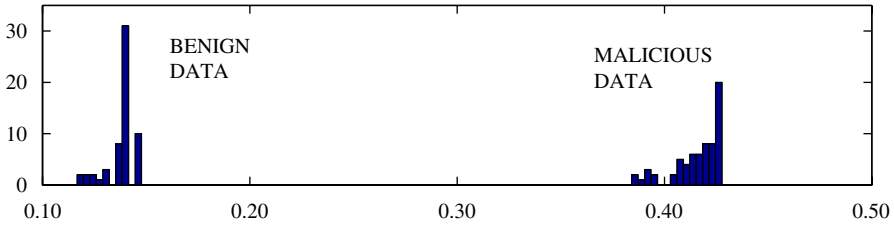

Fig. 4. Histogram of distance from each learning sample to the inferred PST for both benign and malicious application traces.

4.2. Model inference choice

Two approaches to detecting unobserved malicious mobile code exist: signature-based detection and anomaly detection. Signature-based detection involves knowledge of previously observed behavior or elements thereof that constitute malicious activity. The presence of such behavior in a given trace will trigger a detection, whereas an absence will not. Anomaly detection involves learning from benign (i.e., non-malicious) application behavior. If any behavior occurs within a given application that does not correlate with the model to some degree, a detection is triggered. There are benefits and drawbacks to both approaches. Signature-based detection is quite successful in detecting threats provided that the target behavior is known *in advance*. Anomaly detection is successful in detecting *unobserved* behavior. Signature-based detection will not likely produce false positives, i.e., mislabel benign applications as potentially malicious. Anomaly detection, on the other hand, will likely do just that unless the inferred model truly captures all possible benign behavior.

To determine which approach would apply best to the given data, traces from the benign and malicious groups were selected for PST inference and subsequent analysis. The entire population of traces was not used as some applications were represented by multiple code traces whereas others had only a single representative trace. To eliminate this potential bias, a more balanced sampling was chosen by using only one trace from a given application (i.e., no duplicate programs).

An informative measure about patterns inferred is a histogram of the distances from the PST corresponding to each individual sample to the PST inferred from all the samples. The histograms in Fig. 4 show a much smaller distance from the benign dataset to its model T_B than from the malicious dataset to its model T_M. This tighter interval of values for T_B and large separation between the two plots indicates a higher concentration of probabilities in recurrent patterns in the benign application data as compared to the malicious application data. The higher degree of "self-consistency" within the benign application dataset indicates that an anomaly detection approach is appropriate. The reader should note that we are not proposing that the anomaly detection approach is the best choice in all situations. Ultimately, our methods would be used in conjunction with existing frontline defenses (e.g., heuristics and checksums) and signature-based detection schemes.

4.3. Match probability smoothing

A PSA model of benign behavior was inferred using five traces each of Microsoft Word, Microsoft Excel, Microsoft PowerPoint, Microsoft Outlook, and Microsoft Internet Explorer. The set of benign application test data consists of 25 traces from the above five applications in addition to 32 other common applications. A set of 150 malicious application traces from the WildList was also constructed. The motivation behind having multiple (yet different) traces of a given application for model inference is to provide a greater degree of self-consistency. In other words, although different runs of Microsoft Word contain variations on many levels, the underlying application is the same and thus some subset of behavior should exist across all traces generated from Microsoft Word.

Each of the benign and malicious application testing traces was matched against the inferred PSA using the algorithm described in Section 3.4.

Figs. 5 and 6 show the first 100 probabilities of match obtained from a trace of Microsoft Word and an instance of the mass-mailer threat W32.Beagle.K, respectively. The amount of oscillation in probabilities creates difficulty in selecting a proper detection threshold (i.e., the specific probability below which a given trace can be considered malicious). To address this issue, exponentially weighted smoothing was applied to the sequence of probabilities:

$$y_n = \lambda x_n + (1-\lambda) y_{n-1}, \quad 0 \leqslant \lambda \leqslant 1.$$

Values of λ close to one place less emphasis on previous input values thereby causing less smoothing. On the other hand, values of λ close to zero place more emphasis on previous input values thereby causing a greater degree of smoothing.

Figs. 7 and 8 show the same probabilities of match smoothed using $\lambda = 0.1$. This alternate view of the data reduces the oscillatory behavior considerably such that the overall trend of the probabilities is more evident.

4.4. Classification results

Each of the 62 benign application test traces and 150 malicious application test traces was matched against the inferred model. Table 1 shows the number of false positives (i.e., benign applications classified as malicious) and false negatives (i.e., malicious applications not detected). The false positives are divided into Type A, which are test traces from one of the

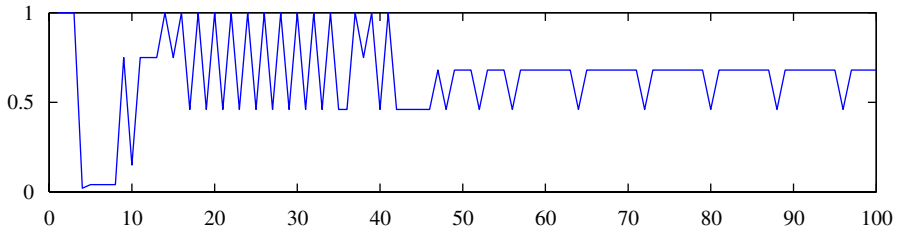

Fig. 5. First 100 API calls from a trace of Microsoft Word.

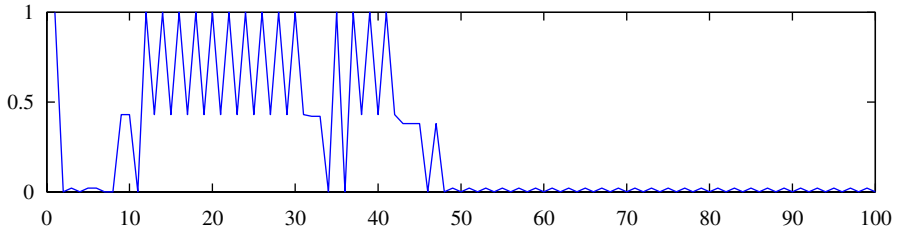

Fig. 6. First 100 API calls from a trace of the worm W32.Beagle.K.

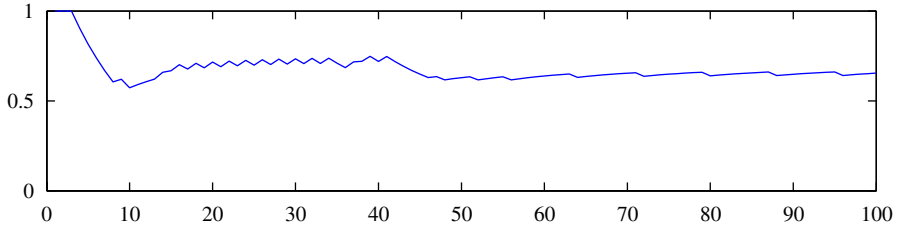

Fig. 7. Data from Fig. 5 smoothed using $\lambda = 0.1$.

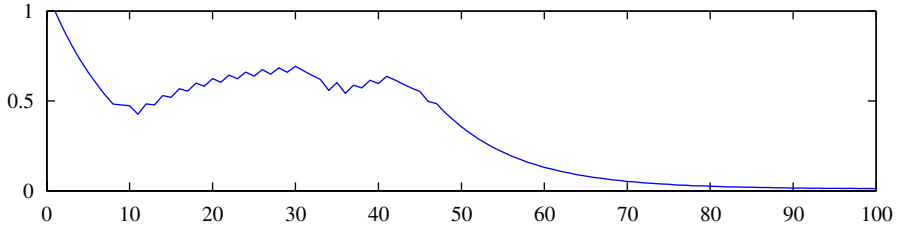

Fig. 8. Data from Fig. 6 smoothed using $\lambda = 0.1$.

Table 1
Model statistics and classification results

Model order	Number of states	False positives		False negatives
		Type A	Type B	
0	1	2	2	14
1	86	1	3	8
2	7135	0	5	6
3	55 047	0	5	6
4	58 924	0	5	6

five Microsoft applications used to build the model, and Type B, which are test traces from the other benign Microsoft and non-Microsoft applications.

First of all, we see that for second- and higher-order models, we correctly identify Type A test traces. That is, we correctly recognize test traces of the same type as those used to build the model. A handful of Type B traces remain incorrectly identified as potentially suspicious regardless of the model order. However, investigation of each of these misclassified benign traces revealed that they all belong to applications that exhibit behavior that is quite different from the training data. One trace represented a run of WinCVS (a version management application) that accessed parts of the file system not accessed by any of the learning samples. Another trace represented a run of AOL Instant Messenger that exhibited many more network-based system calls than any of the learning samples. To properly recognize samples such as these as

benign, examples thereof would have to be added to the training data.

The traces that produced false negatives for second- and higher-order models were also analyzed. Although technically representing malicious applications, none of these traces indicated the state-of-the-system being changed. In one instance, for example, the code entered an infinite loop while searching for a specific registry key to overwrite and was ultimately timed out by the monitoring software before any damage was done. In contrast, all test traces that represented truly malicious behavior were correctly identified.

Using a commodity laptop computer equipped with a 2 GHz Intel Centrino CPU and 2 GB of memory, we found that the time required to match a test sample against the PSA model ranged from 2 ms for the shortest to 178 ms for the longest trace. Thus, even our non-optimized prototype code is sufficiently fast to facilitate runtime processing.

4.5. Clustering results

We applied dynamic agglomerative clustering to the benign-only sample set using PST distances as described above. In this instance clustering was performed to determine if the same traces that differed from the suffix model inferred from multiple traces would retain their same measure of dissimilarity with regard to iterative clustering.

The dendrogram in Fig. 9 shows several clusters that form at a relatively high dissimilarity level. Furthermore, the clusters associated with the greatest distance from the model are the same applications that produced false positive classification results. This again indicates that while a PST/PSA model can capture general characteristics of the learning samples, there are limits to how far this inference extends beyond that data. That is, traces that exhibit vastly different behavior must be included in the training data if they subsequently are to be recognized.

4.6. Incremental learning

The effectiveness of sequential merging as a classification mechanism leads naturally to a performance comparison between inference from scratch of increasingly more samples versus incremental learning. The plot given in Fig. 10 provides some perspective with regard to the time required to yield the desired second order model when the 25 learning sample traces are sorted shortest to longest. The computer was the same as the one mentioned above. First, we see that the cost of building a PST is dominated by the cost of looking up the node labels in the red–black tree that we use for bookkeeping. Secondly, we see that the cost of incrementally adding the sample to the existing PST is dominated by the cost of building a PST for that new sample. Finally, we see that building a PST from scratch when a new sample is added to the training set is much more costly than adding a sample to an existing PST. The same trends are seen for models of higher order.

As a side-remark, we found that fewer nodes were pruned from a PST that represented more samples. Consequently, pruned and full PSTs were nearly identical, implying that

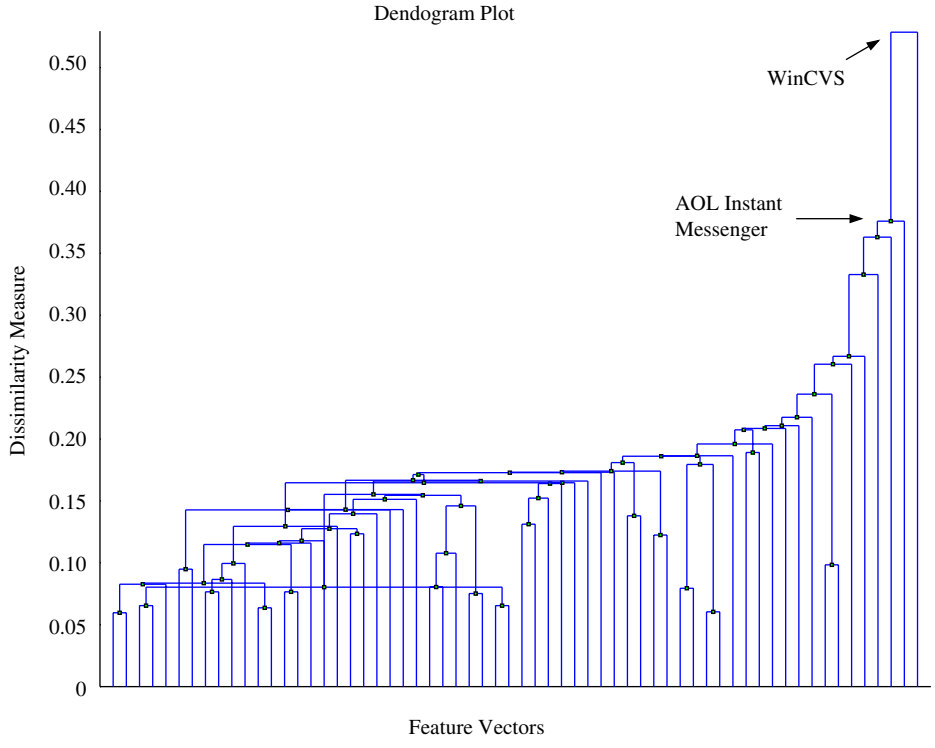

Fig. 9. Dendrogram resulting from agglomerative clustering of benign trace suffix models.

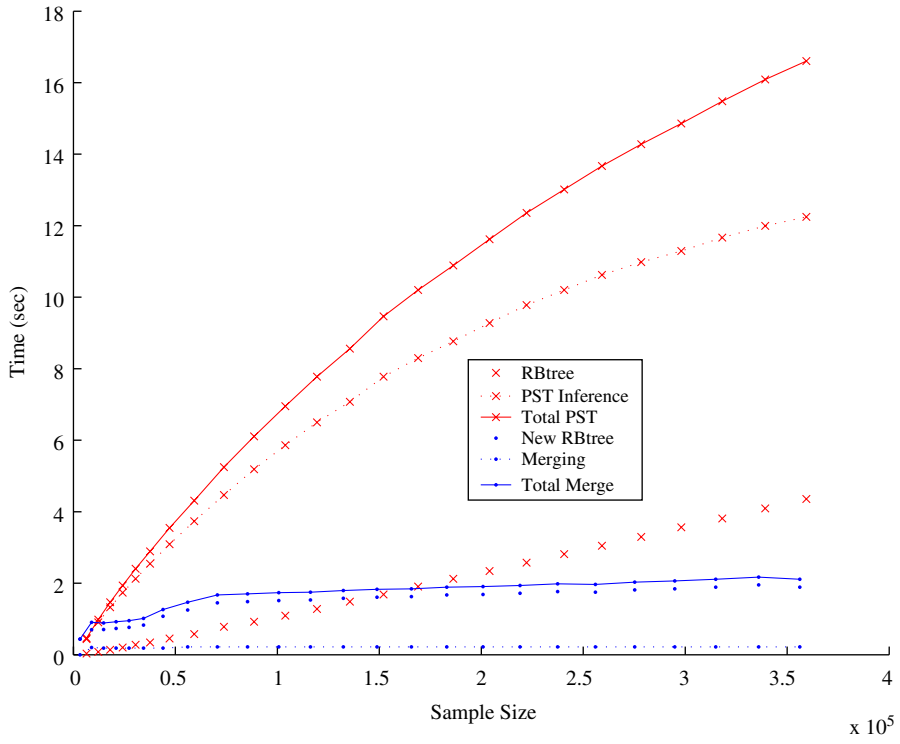

Fig. 10. Timing results for inference a second order PST from scratch (red curves with 'x' marks) versus incremental learning (blue curves with dot marks).

reconstruction of the latter took very little time for the data considered here. Even so, Fig. 10 likely illustrates a generic difference in cost of the two approaches to incorporating new information into a model that carries over to other data for other applications.

5. Conclusion

We have shown that it is feasible to build behavioral, probabilistic suffix models from benign application code traces in the sense that we are able to quite successfully distinguish between code traces representing unknown benign and malicious applications. The small number of false positives and false negatives encountered could all be explained. The false positives, which denote misclassification of benign samples as potentially malicious, were a result of testing a broad set of applications against a model that was inferred from a limited number of samples representing a narrow set of applications. On a more positive note, we successfully recognized a large number of code traces from applications other than those used for model inference. This indicates a certain level of generalization being extracted by the symbolic encoding of the traces and the suffix models. With respect to the false negatives, which denote malicious application code traces that we did not detect as such, then they were found to represent viruses that failed to function correctly in that they essentially terminated without causing damage or replicating themselves.

A full-scale antivirus system would need to be trained on more data. The proposed incremental learning algorithm would be an efficient way to update the suffix model as new training samples become available. Multiple models representing different classes of applications may yield better performance than one large, complex model representing them all. The proposed dynamic clustering algorithm may be a useful tool for determining which applications should be grouped together in the same class. The premise of both of these algorithms is that we can reconstruct the full PST from its pruned version. This in turn was shown to be possible because we only prune nodes that have the exact same next-symbol probability distribution as their parent nodes. Given the number of constraints that can be imposed on the next-symbol frequency count distributions of the collection of PST nodes, it is possible that we might be able to relax the pruning to apply to nodes that are statistically similar to their parents. This remains to be investigated.

Finally, we re-emphasize that the proposed model manipulation, analysis, and usage techniques presented are generic and adaptable to other application areas where machine learning methods can be employed.

Acknowledgment

This work is supported in part by the Office of Naval Research under Grant N00014-01-1-0862.

References

[1] P. Szor, The Art of Computer Virus Research and Defense, Symantec Press, 2005.

[2] E. Skoudis, Malware: Fighting Malicious Code, Prentice-Hall Professional Technical Reference, New York, 2004.

[3] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, N. Weaver, The spread of the Sapphire/Slammer worm, Technical Report, Cooperative Association for Internet Data Analysis (CAIDA), 2003.
[4] L.A. Gordon, M.P. Loeb, W. Lucyshyn, R. Richardson, 2005 CSI/FBI computer crime and security survey, Computer Security Institute Publications, 2005.
[5] S.A. Hofmeyr, S. Forrest, A. Somayaji, Intrusion detection using sequences of system calls, J. Comput. Secur. 6 (1998) 151–180.
[6] S. Forrest, S.A. Hofmeyr, A. Somayaji, T. Longstaff, A sense of self for Unix processes, in: Proceedings of the IEEE Symposium on Research in Security and Privacy, Los Alamitos, CA, 1996, 120–128.
[7] M. Schonlau, W. DuMouchel, W. Ju, A. Karr, M. Theus, Y. Vardi, Computer intrusion: detecting masquerades, Statist. Sci. 16 (2001) 1–17.
[8] M.G. Schultz, E. Eskin, E. Zadok, S.J. Stolfo, Data mining methods for detection of new malicious executables, in: IEEE Symposium on Security and Privacy 2001, Oakland, CA, 2001, pp. 38–49.
[9] L. Feng, X. Guan, S. Guo, Y. Gao, P. Liu, Predicting the intrusion intentions by observing system call sequences, Comput. Secur. 23 (2004) 241–252.
[10] G. Giacinto, F. Roli, L. Didaci, Fusion of multiple classifiers for intrusion detection in computer networks, Pattern Recognition Lett. 24 (2003) 1795–1803.
[11] W. DuMouchel, M. Schonlau, A comparison of test statistics for computer intrusion detection based on principal component regression of transition probabilities, in: Proceedings of the 30th Symposium on the Interface: Computing Science and Statistics, Minneapolis, MN, 2000, p. 404–413.
[12] J. Ryan, M. Lin, R. Miikkulainen, Intrusion detection with neural networks, in: Advances in Neural Information Processing Systems, vol. 10, 1998, pp. 943–949.
[13] G. Tesauro, J.O. Kephart, G.B. Sorkin, Neural networks for computer virus recognition, IEEE Expert 11 (1996) 5–6.
[14] W. Ju, Y. Vardi, A hybrid high-order Markov chain model for computer intrusion detection, J. Comput. Graphical Stat. 10 (2001) 277–294.
[15] D. Yeung, Y. Ding, Host-based intrusion detection using dynamic and static behavioral models, Pattern Recognition 36 (2003) 229–243.
[16] K. Heller, K. Svore, A. Keromytis, S.J. Stolfo, One-class support vector machines for detecting anomalous window registry accesses, in: The Third IEEE Conference Data Mining Workshop on Data Mining for Computer Security, Melbourne, FL, 2003.
[17] J.A. Whittaker, A. de Vivanco, Neutralizing Windows-based malicious mobile code, in: Proceedings of the ACM Symposium on Applied Computing, Madrid, ACM Press, March 2002, pp. 242–246.
[18] R. Ford, M. Wagner, J. Michalske, Gatekeeper II: new approaches to generic virus prevention, in: Proceedings of the International Virus Bulletin Conference, Chicago, IL, 2004.
[19] G. Mazeroff, V. de Cerqueira, J. Gregor, M.G. Thomason, Probabilistic trees and automata for application behavior modeling, in: Proceedings of 43rd ACM Southeast Conference, Savannah, GA, 2003, pp. 435–440.
[20] D. Ron, Y. Singer, N. Tishby, The power of amnesia: learning probabilistic automata with variable memory length, Mach. Learn. 25 (1996) 117–142.
[21] G. Bejerano, G. Yona, Variations on probabilistic suffix trees: statistical modeling and prediction of protein families, Bioinformatics 17 (2001) 23–43.
[22] S. Dubnov, G. Assayag, O. Lartillot, G. Berjerano, Using machine learning methods for musical style learning, IEEE Comput. 36 (2003) 73–80.
[23] T. Cormen, C. Leiserson, R. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1996.
[24] A. Apostolico, G. Bejerano, Optimal amnesic probabilistic automata or how to learn and classify proteins in linear time and space, J. Comput. Biol. 7 (2000) 381–393.
[25] S. Theodoridis, K. Koutroumbas, Pattern Recognition, Academic Press, 1999.
[26] PC viruses in-the-wild, ⟨http://www.wildlist.org⟩, March, 2003.

About the Author—GEOFFREY MAZEROFF received a BA in Music and BS in Computer Science from Furman University, and an MS in Computer Science from the University of Tennessee, Knoxville in 2001 and 2004, respectively. He is currently a candidate for a Ph.D. in Computer Science at the University of Tennessee, focusing on applying probabilistic modeling techniques to areas of computer security.

About the Author—JENS GREGOR was born in Copenhagen, Denmark. He received the MS and Ph.D. degrees in Electrical Engineering from Aalborg University, Denmark in 1988 and 1991, respectively. He then joined the faculty of the University of Tennessee, Knoxville, where he currently holds the rank of Professor of Computer Science. His research interests include syntactic and statistical pattern recognition as well as all aspects of X-ray CT and SPECT image reconstruction.

About the Author—MICHAEL THOMASON received his BS, MS, and Ph.D. degrees, respectively, from Clemson University in 1965, Johns Hopkins University in 1970, and Duke University in 1973. After working at Westinghouse Defense and Space Center, he joined the faculty of the University of Tennessee, Knoxville, where he is a Professor of Computer Science. His research interests include syntactic and statistical pattern recognition, image processing and analysis, and probability models in computer science.

About the Author—RICHARD FORD was born in England, and received a BA, MA and D.Phil. in Physics from the University of Oxford. He is currently on the faculty of Florida Institute of Technology, where he is the Director of the Center for Security Science and an Associate Professor in the Department of Computer Sciences. Research interests center upon security metrics and malicious code detection/prevention.