Probabilistic Trees and Automata for Application Behavior Modeling

Geoff Mazeroff mazeroff@cs.utk.edu
Victor De Cerqueira cerqueir@cs.utk.edu
Jens Gregor jgregor@cs.utk.edu
Michael G. Thomason thomason@cs.utk.edu

Department of Computer Science
University of Tennessee
1122 Volunteer Blvd., Suite 203
Knoxville, TN 37996-3450

ABSTRACT

We describe methods for inferring and using probabilistic models that capture characteristic pattern structures that may exist in symbolic data sequences. Our emphasis is on modeling the sequence of system calls made during the execution of a software application. To obtain learning data, sequences of predetermined system calls are intercepted and mapped into simple symbolic descriptions for a number of runs of the application being modeled. We then construct a probabilistic suffix tree (PST) that represents the probability of each system call given a finite-length sequence of previously observed system calls. Ultimately, we translate the PST into an analogous probabilistic suffix automaton (PSA) which is a parsimonious, variable-order Markov chain. Either model can subsequently be used for real-time data monitoring by means of a matching algorithm. New contributions include an algorithm for comparing two PSTs. We also outline how to extract important Markovian statistics from a PSA and discuss their possible use. Preliminary experimental results are presented based on Visual Basic macros embedded in Microsoft Excel worksheets.

1. INTRODUCTION

In this paper we describe methods for inferring and using probabilistic models based on sequences of software application system calls. Although we concentrate here on application behavior modeling, the methods presented are general by nature and can be used in other contexts.

The premise of our approach is as follows. As an application executes, resource requests are repeatedly made to the operating system in order to read or write a file, allocate memory, dynamically link with a runtime library, create a child process, and so forth. We contend that non-trivial applications exhibit patterns of behavior in the sense that the sequence of system calls associated with the resource requests tend to recur in an order that can be captured by a stochastic model.

Our focus is on malicious mobile code neutralization [8]. In particular, we aim to "fingerprint" the normal behavior of Microsoft Office applications with the goal of later being able to detect abnormal behavior as something that deviates therefrom. As our core models, we have chosen to use the probabilistic suffix tree (PST) and probabilistic suffix automaton (PSA) developed by Ron et al. [6]. Compared with traditional Markov chains, these models allow high-order information to be handled in an efficient way.

Related work includes the general literature on intrusion detection. In addition to malicious mobile code such as email viruses, intrusions come in many shapes and forms including spoofing where a user impersonates another user, software exploitation which often is based on controlled buffer overflow, and packet flooding as seen in denial-of-service attacks. Much of this literature focuses on UNIX application audit trails, e.g., [1] [3] [4] [5] [7], which leads to behavior modeling at a higher level from what is considered here.

The paper is organized as follows. In Section 2 we describe how a Microsoft Office application can be monitored in order to capture the sequence of system calls made during its execution. We describe the PST and its inference in Section 3 together with a new contribution in the form of a method for comparing two PSTs. Cross-model comparisons are a useful tool for determining the extent to which one application model differs from another. In Section 4 we present the PSA and its inference. We also propose methods for application behavior analysis based on match probabilities and Markovian statistics. Finally, Section 5 provides preliminary experimental results based on Visual Basic macros embedded in Microsoft Excel worksheets.

2. APPLICATION MONITORING

The utility we used to capture the sequence of system calls, IMP (Identify/Monitor/Protect), was developed by our research associates at the Florida Institute of Technology [8]. The IMP log provides several pieces of information, such as the call stack, function class (section name), function name, parameters to the function, etc. The following IMP log excerpt describes one function call, `ReadFile`, made during the execution of Microsoft Excel. The notable components of the log entry are the section name, `FILE`,

and the function name, `ReadFile`. These allow us to determine the type of function used as well as its full name.

```
2884:LoggerDll.dll#LoggerDll.dll#
LoggerDll.dll#LoggerDll.dll#OLEAUT32.dll:
12/17/2002  11:53:34:536:FILE:Kernel32.dll:
ReadFile:0:TRUE:3dc:hFile =
C:\WINDOWS\System32\wshom.ocx:13e6a0:
lpBuffer:10:nNumberOfBytesToRead:13e67c:
lpNumberOfBytesRead:0:lpOverlapped
```

Once the log has been generated, we parse it to determine the section and function names of every system call. In order to best correlate the two qualifiers, the function name is appended to the section name (e.g., `FILE_ReadFile`). This correlation will hereafter be referred to as a "log symbol." After parsing the log file we have a list of all unique log symbols, which are then given a one-to-one integer mapping simply for convenient referencing. For example, `FILE_ReadFile` and `REGISTRY_RegOpenKeyExW` may respectively correspond to the numbers 12 and 7 such that

```
...FILE...ReadFile...
...REGISTRY...RegOpenKeyExW...
...FILE...ReadFile...
```

ultimately is represented by the integer sequence

```
12
7
12.
```

This integer encoded system call information is then used as a learning sample for the probabilistic suffix tree.

3. PROBABILISTIC SUFFIX TREES

In the following subsections, we describe the suffix tree (PST) model and how it is inferred from the given learning sample. With respect to model usage, we explain how samples can be matched against PSTs and how PSTs can be compared to one another.

3.1 The Suffix Tree Model

A probabilistic suffix tree is an n-ary tree whose nodes are defined by the occurrence of symbols in the learning sample. The nodal relationship (i.e., child/parent) is based on the parent being a suffix of its children. Each node also contains the probability of symbols that follow the given symbol in the learning sample. An advantage of the suffix tree model is that it is parsimonious. The tree order (depth) is kept to a minimum by excluding nodes that do not provide stochastic information not already given by existing nodes. This process of keeping the tree "trimmed" is described in more detail below. If the tree were not parsimonious, each branch would grow to depth n (where n is the number of symbols) which would make the model impractical to use in a computational sense for large learning samples.

The suffix tree model is best described using an example. Figure 1 is a pictorial representation of a small suffix tree inferred from the learning sample "`1 2 3 1 2 3 2 1 3`".

This model is called a *suffix tree* because each node's parent is a suffix of that node. The "prefix" in this case represents the history of symbols observed thus far. For example, the node "1_2" is a child of "2" because the symbol "2" is a suffix of "1_2". The

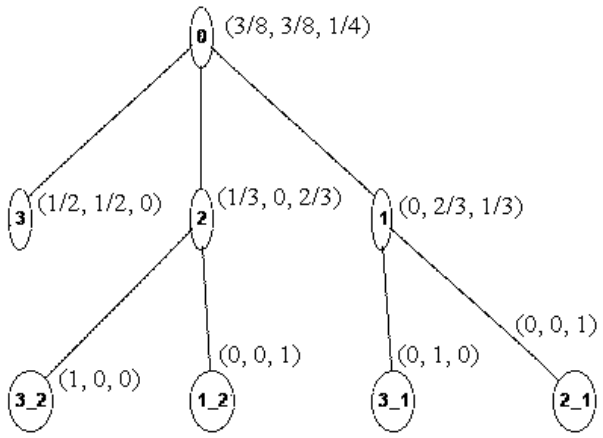

Figure 1: A small probabilistic suffix tree

node "1_2" indicates that the suffix "2" was preceded by "1" in the learning sample. The numbers in parentheses adjacent to the nodes in Figure 1 are the next-symbol conditional probabilities. For example, according to the next-symbol frequency distribution of the node "2", the symbol "2" is followed by "1" a third of the time and followed by "3" the remainder of the time. Observing the learning sample verifies this statement. The root of the PST is the empty symbol, which is inherently a suffix of all other nodes. Note that the probability of next symbol for the root is not the same as the probability of occurrence of each of the individual symbols. If this were the case, the root's next-symbol probability distribution would be $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ since each symbol is equally likely to occur in the sample. This alternate distribution occurs because it is not known which symbol would occur after the last symbol in the learning sample. In this case, it is impossible to infer which of the three symbols would follow the last symbol, "3", in the above sample.

3.2 Inferring the Suffix Tree

The algorithm for building a suffix tree is relatively simple. We start with the root node (empty symbol), and scan through the sample to determine the probability of each symbol occurring. As mentioned above we must adjust the probability distribution to account for the last symbol in the sample. Next, each symbol becomes a child node of the root. For each leaf node in the tree, we scan through the sample to determine the next-symbol probability distribution. During the scan we also take note of which symbols precede the current leaf node because these symbols may become children of the current node. For example if we are currently dealing with the node "1_2" and the sample contains "3 1 2", "3_1_2" becomes a child node of "1_2" because "1_2" is a *suffix* of "3_1_2". If the newly-created leaf node has the same next-symbol probability distribution as its parent, we remove the leaf node. This is done because adding a node with the same probabilities as its parent does not add any additional stochastic information. It is this condition that keeps the tree as parsimonious as the learning data allows. This process of adding leaf nodes continues until no more leaf nodes can be added to the suffix tree.

To better illustrate the effect of "trimming" the tree, note that the sample given in Section 3.1 has the two occurrences of "2 3": one instance followed by a "1" and the other followed by a "2". The reason the node "2_3" does not appear in the tree is because it has the same probability distribution as its parent (i.e., $(\frac{1}{2}, \frac{1}{2}, 0)$). See

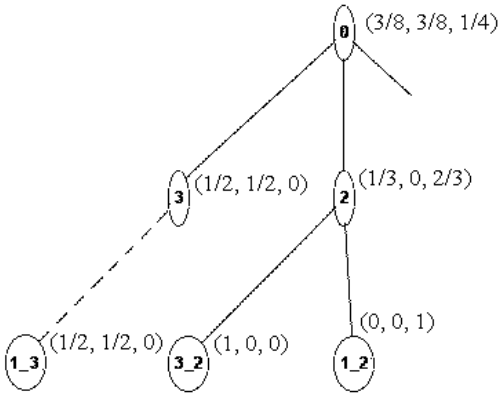

Figure 2: Trimming nodes from the PST

Figure 2.

3.3 Matching Samples Against a PST

The suffix tree can be used as a model against which additional samples are matched. Matching in this sense means traversing the sample while using the PST to calculate the cumulative probability of occurrence for the symbols in the sample.

To match a sample against a PST, we start with the first symbol and look at the root node to determine the probability of encountering that symbol. Next we continue by traversing the sample, appending the next symbol onto the previous one. We then attempt to find the node based on the newly created label excluding the last symbol. For example, if we are trying to match "1_3_2", we attempt to find the node "1_3" because it contains the probability of "2" occurring after it. If the node does not exist, however, we remove one symbol at a time from the front of the node label we are trying to find until we encounter a node that does exist in the tree. This process is called "backtracking" since we move up the tree toward the root until we find a valid node in order to continue the matching process. Note that in the worst case we'll have to backtrack to the root because it should contain the next-symbol probability in order to proceed with the matching algorithm. The cumulative probability of match is found by multiplying the probabilities at each match point against the current cumulative probability, which begins at 1.0.

For example if we wish to match the sample "1 3 2" against the PST shown in Figure 1, the following steps would occur. Step 1: the root node gives a probability of 3/8 for encountering a "1". Step 2: node "1" gives a conditional probability of 1/3 for encountering a "3", so our cumulative probability thus far is 3/24. Step 3: node "1_3" does not exist, so we remove a symbol from the front which gives "3". Node "3" gives a probability of 1/2 for encountering a "2". The cumulative probability of match for the sample is 3/48.

Matching is useful when trying to determine if certain sequences are likely to occur. The higher the probability of match, the more likely the given sample may have been used to infer the suffix tree. For example, when matching "1 2" against the PST from Figure 1, the probability of match is 2/8. This can be verified because out of a possible eight times, "1 2" occurs in the learning data twice. In comparison "1 2" is more likely to have been used to infer the suffix tree than "1 3 2" with its cumulative probability of 3/48.

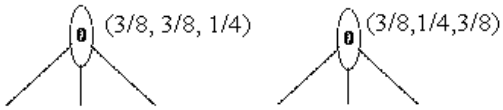

Figure 3: Comparing nodes in the PST

3.4 Comparing PSTs

Comparing two suffix trees can be based on a distance (i.e., amount of difference) between them. The distance between trees is defined as the cumulative distance between the nodes in each tree. The distance between two nodes is defined by a cost metric, for example

$$(x_1 - x_2)^2 + (y_1 - y_2)^2 + ... + (k_1 - k_2)^2$$

is the sum of the squares of the differences. The distance between the two nodes shown in Figure 3 is 0.03125 when using the above cost metric. This distance is zero if and only if the two nodes are identical.

Presently we consider two types of comparisons: semantic and flooded. Semantic comparisons involve comparing only those nodes which exist in both trees. Depending on the learning data, one tree may have many more nodes than the second, making the comparison process more complicated. To address this problem we rely on the fact that if a given node does not exist in the suffix tree, it may have the same probability as its parent. We can "flood" the tree with unnecessary nodes in order to make the comparison by creating as many child nodes as needed while copying the probability distribution from the parent suffix node which exists in both trees. For example, if "1_3" exists in the first tree but not in the second tree, we can "flood" the common suffix node, "3", in the second tree by copying the probability distribution into a new node labeled "1_3". Now that both trees contain "1_3", the cost metric can be applied successfully.

In addition to being used for sample matching and comparative purposes, the PST is used for the inference of an analogous probabilistic suffix automaton.

4. PROBABILISTIC SUFFIX AUTOMATA

In the following subsections, we describe the suffix automaton (PSA) model and how it is inferred from the given suffix tree. With respect to model usage, we explain how samples can be matched against PSAs and how their Markovian statistics can be used in a behavioral sense.

4.1 The Suffix Automaton Model

The suffix automaton is a recurrent discrete-parameter Markov chain of variable-order. The PSA has a well-defined state space and transition function, which gives us the ability to calculate Markovian properties such as steady-state distributions and mean transitions matrices. Another advantage of using the suffix automaton is the ability to generate sequences based on the learning data from which the model was inferred. The concept of a PSA generating a sequence is analogous to the sequence's existence in the learning sample used to infer the suffix tree.

As with the suffix tree, the PSA is best explained using a pictorial example. Figure 4 shows the PSA inferred from the PST in Figure 1. The example PSA contains three transient states (i.e., "0", "1", and "2") and five recurrent states. The values on the arcs connecting

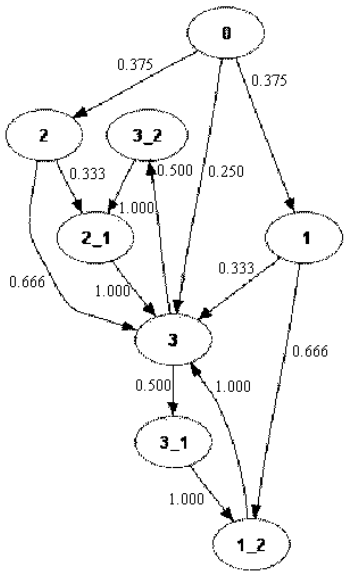

Figure 4: Probabilistic suffix automaton

two given states represents the probability of transitioning from one state to the next. Transitions may only occur where arcs are present. For example, "1" cannot be directly followed by "3_2" because an arc does not exist between them.

4.2 Inferring the Suffix Automaton

Before describing the algorithm for inferring the suffix automaton, it is necessary to note that the nodal relationship in the automaton is slightly different from the relationship in the suffix tree. Instead of parents we have *predecessors* and instead of children we have *successors*. The concept of a predecessor is based on the excluding the right-most symbol. For example, the predecessor to the state "1_3_2" is "1_3", as opposed to the parent of "1_3_2" being "3_2" in the suffix tree.

The suffix automaton is created entirely from the information provided in the suffix tree without any direct reference to the learning sample. The first step is to add all of the leaf nodes in the suffix tree to the automaton as recurrent states. The second step is to create a state that corresponds to the root 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", we create two intermediate states "1_3" and "3" connected from the root state to "3" to "1_3" and finally to "1_3_2". The third step is to create any necessary arcs between states. This 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, we append the symbol to the end of the state label and remove symbols from the *front* of the new label until we find a state that exists in the automaton. For example, state "3_2" in Figure 4 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". The final step is to assign state types to the states created in Steps 2 and 3. This is done by visiting each of the created nodes and determining if any edges incident upon them originated from recurrent states. If this is the case, the given node 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 nodes not marked recurrent after visiting each node are defined as transient by default.

4.3 Matching Samples Against a PSA

Matching samples against a suffix automaton yields the cumulative probability of the given sample being generated by the PSA.

To match a sample against a PSA, we start with the first symbol and find its corresponding state in the automaton. Next we continue by traversing the sample, appending the next symbol onto the previous one. We then attempt to find the state based on the newly created label. If the node exists, we note the probability of transitioning from the previous state to the current state and continue appending symbols from the sample. If the node does not exist, however, we remove one symbol at a time from the front of the state label we are trying to find until we encounter a node that does exist in the automaton. Note that in the worst case we will have to backtrack to a transient node because it should contain the next-symbol probability in order to proceed with the matching algorithm. This worst-case is similar to backtracking to the root node in the PST. For example, if we wish to match the sample "1 3 2" against the PSA in Figure 4, the following steps would occur. Step 1: node "1" exists, so append "3" which occurs with probability 1/3. Step 2: node "3_2" exists and a transition occurs with probability 1/1. The cumulative probability of match is 1/3.

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 sample. For example, the probability of match for the sample "1 3 3" is 1/8, which is less likely than the previous example "1 3 2". The lower probability of match is verified by noting that two adjacent 3's never occur in the learning sample.

4.4 Markovian Statistics

Markov models provide a wide array of descriptive statistics [2]. We mention two such statistics: the steady-state distribution and the expected number of transitions until a given state is reached. The steady-state distribution of a recurrent discrete-parameter Markov chain is defined as

$$\pi_j = \sum_i \pi_i p_{ij}$$

or, in matrix notation, $\Pi = \Pi P$ where P is the transition matrix and Π is a row vector of steady-state probabilities for each recurrent state. The steady-state distribution π_j is the probability of the automaton being in state j in the long-run. This statistic is useful when determining which states are least likely to be encountered over a significantly large sample.

The expected number of transitions until a given state is reached is defined as

$$m_{ij} = 1 + \sum_{k \neq j} p_{ik} m_{kj} .$$

Given that the automaton is in state i, the mean number of transitions m_{ij} is the average number of transitions until the next occurrence of j in the long-run.

As an illustrative example, consider the PSA in Figure 4. We first identify the subset of states that are recurrent, namely, { "1_2", "2_1", "3", "3_1", "3_2" }. The corresponding steady-state distribution is $\Pi = [\frac{1}{6}, \frac{1}{6}, \frac{2}{6}, \frac{1}{6}, \frac{1}{6}]$. The most likely state is thus "3". The other recurrent states are half as likely to occur. With respect to the mean transition matrix then we obtain

$$m = \begin{bmatrix} 6 & 6 & 1 & 5 & 5 \\ 6 & 6 & 1 & 5 & 5 \\ 5 & 5 & 3 & 4 & 4 \\ 1 & 7 & 2 & 6 & 6 \\ 7 & 1 & 2 & 6 & 6 \end{bmatrix}$$

where the ith column or row in the matrix corresponds to the ith recurrent state. Thus, the mean number of transitions from "3" to "3_1" corresponds to $m_{34} = 4$. Additionally, the mean number of transitions from "1_2" to "3" corresponds to $m_{13} = 1$, which is expected since the only transition from state "1_2" is to state "3".

5. BEHAVIOR ANOMALY DETECTION

In the following subsections we describe how the stochastic models defined thus far can be used to detect anomalous behavior in the context of Visual Basic macros embedded in Microsoft Excel worksheets. We provide descriptions of the macros themselves and several examples of the model usage techniques mentioned previously. Specifically we discuss encountering foreign symbols, comparing PST models, matching samples against a benign model, and using Markovian statistics from the PSA.

5.1 Macro Descriptions

In order to demonstrate model usage, five macros were utilized – two of which were considered malicious. Malicious in this sense means that the tasks performed are not typically handled by macros, and could possibly achieve undesired effects on the user's machine. The first macro, `Accumulate`, manipulates numeric values contained in worksheet cells A1 through A30 by doubling their current values. The second macro, `Cellrange`, alters the values in the same range as `Accumulate` except it uses a loop to insert the values 1 through 100 into each of the cells. The third macro, `ABCLight`, creates a graph based on cell values entered by the user. The fourth macro, `Homepage`, modifies the registry key used by Microsoft Internet Explorer which contains the "home page" for the browser. The fifth macro, `Spawnshell`, executes another application from within Excel. The first three macros are considered benign since they simply manipulate cell values within the context of the Excel worksheet. The last two macros are considered malicious since they manipulate the Windows registry and execute separate applications. This behavior is of consequence because the ILOVEYOU virus begins propagating itself by setting the default home page to a specified Web address, and the application being executed by `Spawnshell` may very well be an application that corrupts files on the user's machine [8].

5.2 Foreign Symbols

If during a "malicious" execution a system call is made in a context which did not occur during the benign execution, both the PST and PSA would yield a probability of zero for encountering the new symbol. Therefore, both models can be used to detect new behavior, which in this sense is deemed anomalous. For example, `Spawnshell` is the only macro that invokes the system call `CreateProcess`. Since each function corresponds to a unique symbol, the symbol for `CreateProcess` is not present in any of the other models for the remaining macros. Therefore, this macro could be deemed malicious based on not having seen the new symbol in the modeled context.

5.3 Comparing PSTs Against a Benign Model

The behavior of the individual macros can be used to infer PSTs which can then be compared to determine the distance between them. We want to see whether the distance between PSTs modeling malicious behavior and PSTs modeling benign behavior will be greater than the distance between two PSTs representing only benign behavior. We constructed a benign model using the samples generated for the `ABCLight` and `Accumulate` macros. The table below lists the distance between each of the PSTs generated for the five individual macros and the newly constructed benign model using a direct comparison with an absolute sum of the differences cost metric.

ABCLight	Accumulate	Cellrange	Homepage	Spawnshell
96.457	96.563	148.979	140.902	168.386

The models that differed most from the reference PST correspond to the two malicious macros as well as the benign macro which was not used to infer the benign model. The `Cellrange` macro is considered benign, yet it differs from the benign model more so than the malicious `Homepage` macro. This demonstrates the need to have a more encompassing benign model against which comparisons can be made. Otherwise macros considered to be benign may be deemed malicious when they actually are not (i.e., a false positive).

5.4 Matching Against a Benign Model

As a candidate sample increases in length, matching algorithms can be used "on the fly." Such techniques have been investigated using a "sliding window" approach in order to obtain a series of match probabilities [1]. Ideally a benign sample when matched against a benign model should yield a high probability of match. Likewise, a malicious sample should yield a low probability of match. In reality however, a malicious sample could have a high probability of match except for a few instances when the probability is much lower than expected - which may be indicative of anomalous activity. The examples presented below are based on sparse learning data and may not reflect the results of matching samples against a much larger benign model.

We matched the samples for `Cellrange` and `Homepage` against the benign model described in Section 5.3. In order to properly detect anomalies in this case, the probability of match should be higher for `Cellrange` since it is classified as a benign macro. Figure 5 verifies this behavior since the probability of match averages higher for `Cellrange` than for `Homepage`. Specifically for this sequence, `Cellrange`'s probability of match exceeded `Homepage`'s probability of match 63% of the time. Figure 6 demonstrates that when we compare a sample which was used to create the benign model against the model itself, the mean probability of match is higher. Specifically, the mean probability of match for `Cellrange` and `ABCLight` are 0.242 and 0.247 respectively, whereas the mean probability of match for `Homepage` is 0.165.

5.5 Markovian Statistics

The Markovian statistics made available by the PSA can be used as well. The steady-state distribution describes how likely each state

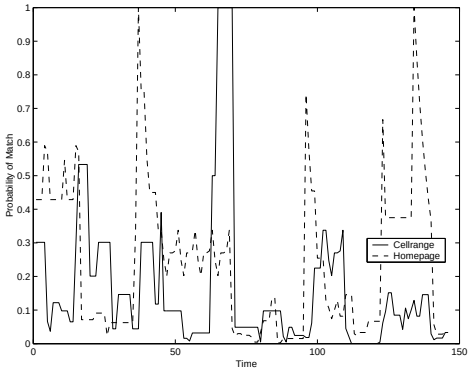

Figure 5: Matching benign and malicious macros

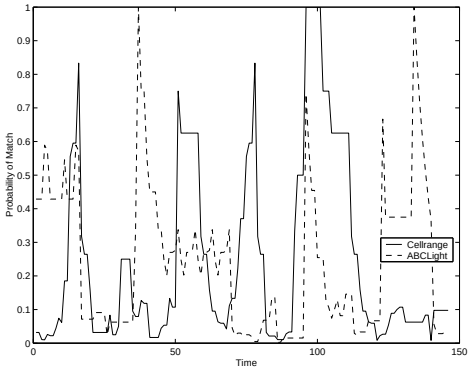

Figure 6: Matching benign macros only

is to be visited in the long-run. If a sample causes the automaton to be in an unlikely state with great frequency, the sample could be considered anomalous. The mean number of transitions is also of interest. If a sample causes the automaton to transition from state i to state j in a consistently low number of transitions and m_{ij} is quite large, this may indicate that the observed sample is getting from state i to state j through some shorter route than the expected one.

With the benign model mentioned previously, the least-likely state ($\pi = 0.000044$) involves the system calls `RegEnumKeyExW`, `RegCloseKey`, `RegCloseKey`, `RegQueryValueExW`, and `RegQueryValueExW` in that order. Conversely, the most-likely state ($\pi = 0.035659$) involves the system calls `RegOpenKey` followed by three repetitions of the sequence `ReqQueryValueExW`, `RegCloseKey`, and `RegCloseKey`. If a PSA were generated to model a given macro and the system calls in new data occurred with greater relative frequency than in the model, the new data could be deemed suspicious. With regard to the mean number of transitions and the least-likely and most-likely states, then we observe the following values:

m_{ij}	Least-likely	Most-likely
Least-likely	22,475	441
Most-likely	22,557	28

Thus, if during execution the least-likely state is accessed from the most-likely state after just a few hundred state transitions then the system call sequence being analyzed is not typical. The same would, of course, be true for other significant state-pair statistics deviations.

6. CONCLUSION

The goal of this paper was to describe a method of obtaining sequences of applications system calls and for building and using stochastic models based on such sequences. We have presented a means of capturing and encoding system call information for Microsoft Windows-based applications using IMP. We have also presented two stochastic models, the probabilistic suffix tree and probabilistic suffix automaton, as a means of modeling sequential samples. We have demonstrated that the models can serve as behavior profiles in order to detect observed anomalous behavior. We believe that these models can ultimately be used to profile benign behavior in a target application (e.g., Microsoft Excel) so that malicious behavior (e.g., macro viruses) can be detected. With this concept in mind, the probabilistic models described in this paper may be of use in the realm of computing security as a means of detecting malicious application behavior.

7. ACKNOWLEDGEMENTS

The work is supported by the Office of Naval Research under grant number N00014-01-1-0862. Portions of the figures in this paper were generated by the software package SPARTA as described in a companion paper for the 2003 ACM Southeast Conference.

8. REFERENCES

[1] S.A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[2] J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. Springer, New York, 1960.

[3] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, Orlando, FL, 1994. IEEE Computer Society Press.

[4] T. Lane and C.E. Brodely. Temporal sequence learning and data reduction for anomaly detection. *ACM Trans. Information and System Security*, 2:295–331, 1999.

[5] T.F. Lunt. Detecting Intruders in Computer Systems. In *Proceedings of the Sixth Annual Symposium and Technical Displays on Physical and Electronic Security*, 1990.

[6] D. Ron, Y. Singer, and N. Tishby. The power of amnesia: Learning probabilisitc automata with variable memory length. *Machine Learning*, 25:117–142, 1996.

[7] M. Schonlau, W. DuMouchel, W. Ju, A. Karr, M. Theus, and Y. Vardi. Computer intrusion: Detecting masquerades. *Statistical Science*, 16:1–17, 2001.

[8] J.A. Whittaker and A. De Vivanco. Neutralizing Windows-based malicious mobile code. In *Proc. ACM Symposium on Applied Computing (Madrid, March 2002)*, pages 242–246. ACM Press, 2002.