

To the Graduate Council:

I am submitting herewith a thesis written by Victor P. De Cerqueira entitled: "SPARTA: A Graphical User Interface for Malicious Mobile Code Fingerprinting." I have examined the final electronic version of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Jens Gregor,
Major Professor

We have read this thesis
And recommend its acceptance:

Michael G. Thomason

Bradley Vander Zanden

Accepted for the Council:

Anne Mayhew
Vice Provost and Dean of Graduate Studies

(Original signatures are on file with official student records.)

SPARTA:
A Graphical User Interface for
Malicious Mobile Code Fingerprinting

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Victor Pimentel de Cerqueira

May 2003

Acknowledgement

First of all, I would like to thank my advisors, Drs. Michael Thomason and Jens Gregor for their continual guidance and keen insights throughout the research work that resulted in this thesis. I would also like to thank Dr. Bradley Vander Zanden for his valuable help as a GUI expert, and for serving on my thesis committee. I am also grateful to my research partner, Geoffrey Mazeroff, for making my work so enjoyable.

This research would not be possible at all without the funding provided by the Office of Naval Research, under the grant N00014-01-1-0862.

Outside the academic realm, I am overwhelmed with thankfulness to my parents, who encouraged me to pursue a degree in the U.S.A. and provided the means for that end, despite the financial strain it meant at first. Above all, I am grateful to God, who blessed me with health, peace, and courage to overcome my limitations and insecurities during my life in academia.

Abstract

This thesis introduces and describes SPARTA (for Stochastic Profiling Application for the Rendering of Trees and Automata), a graphical user interface used as a front end to a collection of tools written in C that collectively convert a log of registry system calls performed by an application into binary descriptions of PSTs (for Probabilistic Suffix Trees) and PSAs (for Probabilistic Suffix Automata), which are models used to represent application behavior on Windows-based systems. SPARTA works by rendering these binary descriptions into graphical form, showcasing a variety of features intended to make the user interaction with PSTs and PSAs informative and insightful.

The ultimate goal of SPARTA is to aid in the process of profiling applications based on the system calls they make, using characteristics from PSTs and PSAs that are more easily noticeable in their graphical form to define “normal” behavior for Windows applications. With knowledge of normal behavior, these very same models can be used to measure deviations that might ultimately result in the destructive actions of malicious mobile code, enabling the user to halt or quarantine them before they take place.

Contents

Chapter	Page
1. Introduction	1
1.1 Motivation and background	1
1.2 The Conception of SPARTA	4
1.3 Goals and Overview	5
 2. The Tools Available in SPARTA	 8
2.1 Introduction	8
2.2 The Symbol Mapper	9
2.3 Probabilistic Suffix Trees	11
2.3.1 Introduction	11
2.3.2 PST Builder	12
2.3.3 PST Comparator	13
2.3.4 PST Matcher	14
2.3.5 PST Merger	15
2.3.6 PST Viewer	16
2.4 Probabilistic Suffix Automata	25
2.4.1 Introduction	25
2.4.2 PSA Builder	26

2.4.3 PSA Condenser	28
2.4.4 PSA Matcher	29
2.4.5 PSA Scanner	30
2.4.6 PSA Viewer	30
3. Graph Layout Techniques in SPARTA	38
3.1 Introduction	38
3.2 The Basic Algorithm	40
3.3 Graphplace vs. DOT	42
3.3.1 Graphplace	42
3.3.2 DOT	44
3.3.3. Conclusion	47
4. Design Concerns and Important Algorithms	49
4.1 Java as The Development Platform	49
4.2 The Graph Rendering Scheme	51
4.2.1 X-Y Vectors	53
4.2.2 X-Y Vectors vs. Kd-Trees	55
4.2.3 Edge Drawing Techniques	57
4.2.3.1 PST Edges	57
4.2.3.2 PSA Edges	59

5. Conclusion and Future Work	62
Bibliography	64
Vita	70

List of Figures

Figure	Page
1.1 From ASCII to Java art: a textual description of a PST and its corresponding graphical representation	6
1.2 A textual description of a PSA and its corresponding graphical Representation	7
2.1 SPARTA's main menubar	9
2.2 The registry mapping created by the Symbol Mapper	10
2.3 The PST Comparator Dialog Box	14
2.4 The result of a PST comparison	15
2.5 A PST in "skeleton mode"	20
2.6 A PST with merged nodes	21
2.7 A snapshot of the visible part of a PST	22
2.8 The PST Debugger's monitor	24
2.9 The PST Debugger's breakpoint table	24
2.10 A plot for the cumulative probabilities of match for a sample consisting of 50 symbols partitioned in windows of size of 10	25
2.11 A PSA in "Skeleton Mode"	27
2.12 A super node in a PSA	29

2.13 The PSA Matcher dialog box	31
2.14 The PSA Scanner dialog box	31
2.15 The dialog box for probability range color assignment	34
2.16 A chooser dialog for edge colors	35
2.17 A colored horizontal bar summarizes the probability ranges and their respective color patterns, as established by the user	35
2.18 The visible portion of a PSA with the probability color patterns established by the user	36
2.19 A “backtracked” Node in the PSA Debugger	37
3.1 A PST laid out with Graphplace and a PSA laid out with DOT	48

Chapter 1

Introduction

1.1 Motivation and Background

The Internet is in a constant process of expansion, as the World Wide Web is joined every day by a variety of institutions, ranging from banks offering on-line transactions, through supermarkets seeking to expand their profits in the world of e-business, to government agencies thriving to enhance their services and cut costs with personnel. As a consequence, the flow of data across networks has become more intense - ultimately resulting in more computers being exposed to the threat of destructive programs such as viruses, worms, and trojan horses that often run shrouded from users. These programs are collectively referred to as *malicious mobile code*.

It is a very difficult task to deal with malicious mobile code intrusion at the level of system design, insofar as computer systems are dynamic by nature: any correctly specified security policies or any piece of software following the most rigorous standards of software engineering will most likely be rendered unreliable by the time a vendor implements a new “update” to the system, or a system administrator decides to change configurations, or even an ordinary user de-

cides to install or remove a program. For this reason, a great deal of research effort has been placed in the search of more promising computer security methods. A brief overview and discussion of some of these methods follows.

Monitoring user behavior: The IDES model proposed by Denning[4] monitors user activity in a computer system and builds a profile of normal behavior, in an attempt to identify abnormalities that possibly translate into malicious behavior using statistical methods. The model proposed by Ju and Vardi[28] performs similar work using a high-order Markov chain to profile UNIX command sequences, and has yielded promising results. Nevertheless, monitoring user behavior can be a very insidious task, since the range of behaviors for users can be large (if not unbounded), has great variability, and tends to change over time.

Monitoring system behavior: Two different approaches can be applied to this method. The first approach, known as *misuse detection*, relies on matching the system behavior to databases of known intrusion signatures, which means that new types of attack will most likely go by undetected. The second approach, known as *anomaly detection*, attempts to profile normal behavior and flag (possibly new) behaviors that deviate from it. This latter approach is more promising and has been the research focus of Hofmeyr, Forrest, and Somayaji [24], who attempt to monitor short sequences of system calls performed by *privileged processes* (programs that require access to system resources that are inaccessible to the ordinary user, such as sending or receiving mail) in UNIX to infer normal behav-

ior. Warrender, Forrest, and Pearlmutter[2] experiment with several modeling methods to represent normal behavior accurately and to recognize intrusions, including simple enumeration of observed sequences of system calls, comparison of relative frequencies of different sequences, and Hidden Markov Models (HMMs), which are finite state machines whose states represent some unobservable condition of the system being modeled. They conclude that HMMs are the most powerful of all approaches, but come at high computational cost, and suggest that simpler methods may yield acceptable results.

Other methods: Some security analysis tools deal with malicious mobile code by scanning a system for weaknesses and possible security holes that could lead to intrusions, rather than dealing with the intrusions themselves. This is the case of COPS[5] and SATAN[6]. Finally, a popular technique employed by some of the most popular commercial tools is *sandboxing*, which basically consists of specifying a limited and controlled space where code can be executed with a reduced risk of damage. Despite their popularity, sandboxes are ineffective, as tools like Surfinguard and e-Safe failed for even some of the most popular threats such as the “love bug” and the Melissa virus [14].

1.2 The Conception of SPARTA

Following the trend of system behavior monitoring aforementioned, a team of researchers from the Florida Institute of Technology led by Dr. James Whittaker devised a tool called HEAT (for Hostile Environment Application Tester), which has managed to neutralize major Windows-targeted viruses such as the “love bug”, Melissa and PrettyPark [13]. During the monitoring task, HEAT makes judgment calls about malicious behavior using a hybrid approach of misuse and anomaly detection: the misuse part consists of a hard-coded set of known impermissible actions (such as writing specific registry keys), and the anomaly detection part relies on a list of “questionable” behaviors, which requires a far more sophisticated approach based on pattern analysis and data mining. This is where the research of Drs. Jens Gregor and Michael Thomason from the University of Tennessee comes into place, proposing the use of Probabilistic Suffix Trees (PSTs) and Probabilistic Suffix Automata (PSAs) as identifiers of unique signatures for diverse Windows applications. The idea is that known benign programs could be profiled and used to define normal behavior. These models were initially implemented as a collection of tools developed in the C language that in essence convert a raw data file generated by HEAT containing a log of system calls performed by an application to PSTs and PSAs. It soon became obvious that such models were of very limited use in their original ASCII form, insofar as their tex-

tual representation were a far departure from their intrinsically graphical nature, making it difficult to interpret them. To remedy this deficiency, a graphical user interface for these models called SPARTA (Stochastic Application for the Rendering of Trees and Automata) was developed, providing a smooth front end to the output generated by the C binaries, as shown in Figures 1.1 and 1.2.

1.3 Goals and Overview

The goal of this thesis is to describe SPARTA in full detail, showing how its various features can be used in the inspection and manipulation of the stochastic and Markovian models deployed in malicious mobile code detection.

The chapters that follow outline the SPARTA GUI and its applicability in full detail. Chapter 2 provides a complete description of all the tools available in SPARTA, along with compact definitions of PSTs and PSAs. Chapter 3 discusses the graph layout techniques used for rendering PSTs and PSAs, assessing their cost-benefit aspects. Chapter 4 outlines some of the key design concerns during the creation of SPARTA, and discusses the algorithms employed to render the usually large PSTs and PSAs in a smooth and efficient fashion. A summary of this thesis and a discussion of future work are provided in Chapter 5. For the rest of this discussion, PSTs and PSAs will be generically referred to as “graphs” whenever appropriate, since that is the underlying structure in which they are defined in SPARTA.

```

test10_pst_test.txt - Notepad
File Edit Search Help
|5 15 8 2 0 2
1 2 5 6 7 8 12 13
0 7 2 7 0.2916666667 5 4 0.1666666667 6 4 0.1666666667 7 4 0.1666666667 8 2 0.0833333333
13 1 2 2 1.0000000000 0
12 1 2 1 1.0000000000 0
8 1 6 2 1.0000000000 0
7 3 1 1 0.2500000000 7 2 0.5000000000 12 1 0.2500000000 0
7_7 2 1 1 0.5000000000 12 1 0.5000000000 0
5_7 1 7 2 1.0000000000 0
6_2 6 2 0.5000000000 13 2 0.5000000000 0
8_6 1 6 2 1.0000000000 0
6_6 1 13 2 1.0000000000 0
5_2 7 2 0.5000000000 8 2 0.5000000000 0
2_2 2 3 0.4285714286 5 4 0.5714285714 0
13_2 1 2 2 1.0000000000 0
12_2 1 5 1 1.0000000000 0
2_2 1 5 3 1.0000000000 0
5_7_7 2 1 1 0.5000000000 12 1 0.5000000000 0
2_5_2 7 2 0.5000000000 8 2 0.5000000000 0

```

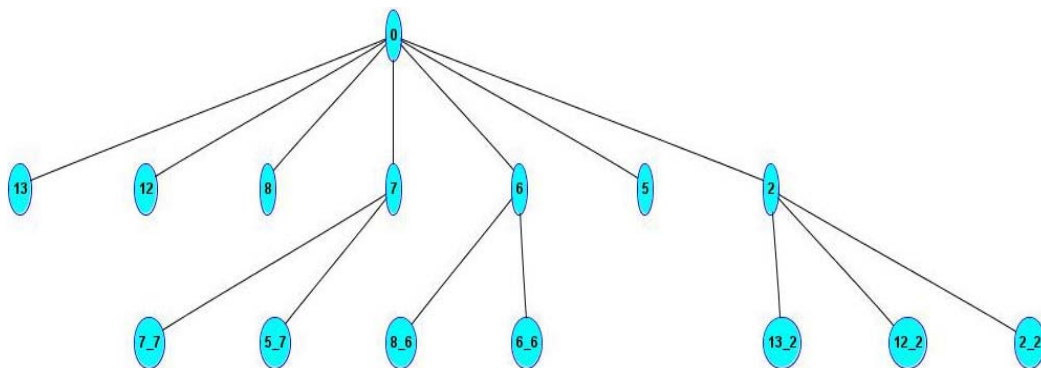


Figure 1.1 – From ASCII to Java art: a textual description of a PST and its corresponding graphical representation.

```

test10_psa_test.txt - Notepad
File Edit Search Help
0 0 : t : ns= 7 np= 0 : 0
    7 9 11 13 15 2 4
    7 4 4 4 2 1 2
    0.291667 0.166667 0.166667
    n
1 1 : r : ns= 1 np= 2 : 1
    6
    2
    1.000000
    n
2 2 : r : ns= 1 np= 3 : 12
    3
    1
    1.000000
    n
3 3 : r : ns= 1 np= 1 : 12_2
    9
    1
    1.000000
    n
4 4 : r : ns= 1 np= 4 : 13
    5
    2
    1.000000
    n
5 5 : r : ns= 1 np= 1 : 13_2
    8
    2
    1.000000
    n
6 6 : r : ns= 2 np= 1 : 1_6
    12 4
    4 4
    0.500000 0.500000
    n
7 7 : t : ns= 2 np= 1 : 2
    8 9
    3 4
    0.428571 0.571429
    n

```

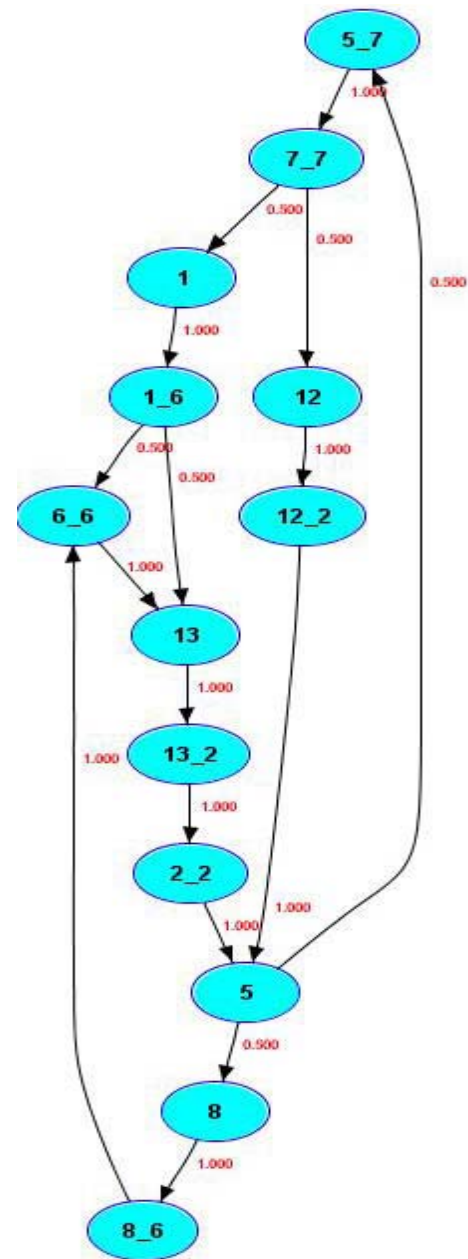


Figure 1.2 – A textual description of a PSA and its corresponding graphical representation.

Chapter 2

The Tools Available in SPARTA

2.1 Introduction

SPARTA consists of a menubar from which all of the PST and PSA-related tools can be accessed (Fig. 2.1). This menubar consists of five menus:

- *SPARTA Options*: This is the main application control menu, and basically allows the user to exit SPARTA and all of its sub-applications altogether;
- *PST Actions*: This menu contains tools for PST building, viewing, comparing, merging, and sample-matching;
- *PSA Actions*: This menu contains tools for PSA building, viewing, condensing, scanning, and sample-matching;
- *Other Tools*: Currently consists of one single tool for sample symbol mapping;
- *Help*: Provides information about the SPARTA team and funding source.

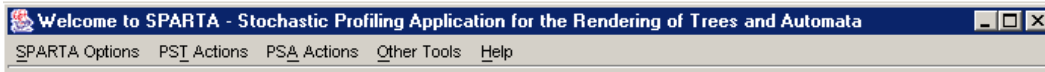


Figure 2.1 – SPARTA’s main menubar.

2.2 The Symbol Mapper

A Windows application’s behavior can be identified by the sequence of system calls the application performs during its execution. Different runs of an application are likely to generate different sequences of system calls, since the user might not always repeat the same set of actions within the program at each run. Nevertheless, there is a great chance that after a certain number of runs, the combined sequences of system calls will yield what can be defined as "normal behavior" for a given application.

In order to model a Windows application by a tree or an automaton model, it is imperative that its system calls are represented in an efficient, concise fashion, allowing for easy manipulation and aesthetically pleasing display. Hence the need for SPARTA's first tool: the Symbol Mapper. Located in the menu *Other Tools*, this is the first application to be run before a PST or PSA can be inferred for an application. It takes as input a log file consisting of raw data generated by a monitoring tool for Windows applications which traces, among other things, the

registry system calls performed by these applications during an execution cycle. As output, the Symbol Mapper generates a *.map* ASCII file mapping registry entries to integer symbols, and a *.symb* ASCII file containing the registry calls performed by the application, properly translated into the symbols from the *.map* file (see Fig. 2.2). The *.symb* file serves as input to the PST Builder, which is described in the next section.

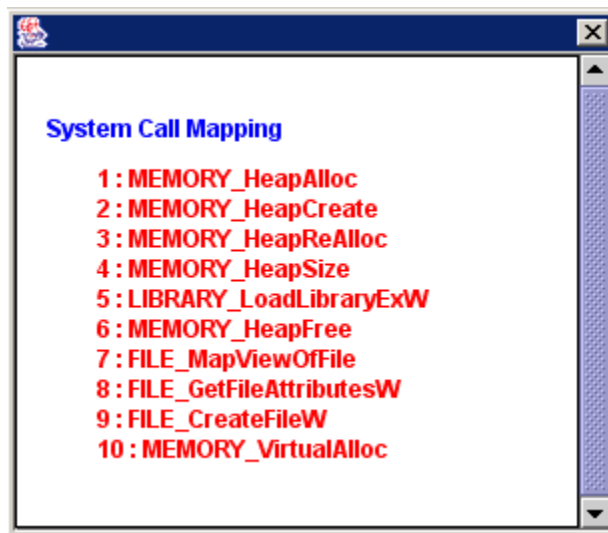


Figure 2.2 – The registry mapping created by the Symbol Mapper.

2.3 Probabilistic Suffix Trees

2.3.1 Introduction

A Probabilistic Suffix Tree (PST) over a given alphabet Σ is a non-empty tree where:

- The degrees of nodes range from zero to the size of the alphabet;
- Each edge in the tree is labeled by a single symbol from the alphabet;
- A given node in the tree is labeled by a string constructed by concatenating the labels from the edges in the path from that node to the root;
- Associated with each node is a conditional probability vector of the same size as the alphabet. This vector describes the probability distribution on the next symbol appearing right after the string labeling that node.

This kind of data structure is suitable for making predictions. The likelihood of a sequence S of symbols by a certain PST is performed symbol by symbol, where the probability of each symbol is calculated by scanning the tree to search for the longest subsequence that appears in the tree and ends just before that symbol. The conditional probability of this symbol given this suffix is given by the probability

distribution associated with the corresponding node in the PST. The reader is encouraged to read [20] for a more detailed discussion of PSTs.

The next three subsections discuss in the detail the steps necessary for building and visualizing a PST, using the tools from the *PST Actions* menu in the SPARTA suite.

2.3.2 PST Builder

Once the sample represented by the *.symb* file containing the sequence of registry calls performed by an application properly mapped into integer symbols is created by the Symbol Mapper, a PST can be constructed so as to represent a "stochastic picture" of the application in question. This is where the PST Builder comes into place. It reads in the sample and generates a *.pst* binary file, which can then be viewed by the PST Viewer (which is described in the next subsection). Before creating the PST, the PST Builder prompts the user for arguments, namely the number of symbols to be used, the maximum order allowed for the tree, and a threshold value for the number of nodes added to the tree. It also allows the user to decide whether a red-black tree (*.rbt*) file containing information on all possible node labels in the sample and how often they occur should be created. This red-black tree can then be used to quickly generate the PST, keeping the PST Builder from having to perform multiple linear scans through the sample. Furthermore, the *.rbt* file can be used with the PSA Builder (described in the next section) so

that frequency counts can be properly "installed" as nodes are added to the automaton and links between successors and predecessors are established.

Once the builder is finished constructing the PST, the user is prompted for instantaneous visualization of the tree.

2.3.3 PST Comparator

As the name suggests, the PST Comparator is a tool for comparing trees. It compares two trees at a time and operates on two modes, *direct* or *flooded*, which can be specified by the user. A direct comparison means that only nodes that are common to both trees are accounted for during the comparison; a flooded comparison first adjusts both trees so that they share the same structure, allowing all possible nodes to be used in the comparison. In either case, the comparison is performed by calculating the distance between the two trees, which is the cumulative distance between the nodes in each tree. In SPARTA, the distance between two nodes can be specified by one of the following cost metrics: the *absolute sum of the differences* or the *sum of the squares of differences*. The PST Comparator makes all these settings available on a dialog box (see Fig. 2.3) which also allows the user to specify (either by typing or by means of a file chooser) the files containing the PSTs to be compared; every time the *Compare* button is pressed, the *pst_compare.exe* binary is executed in the background with the specified options

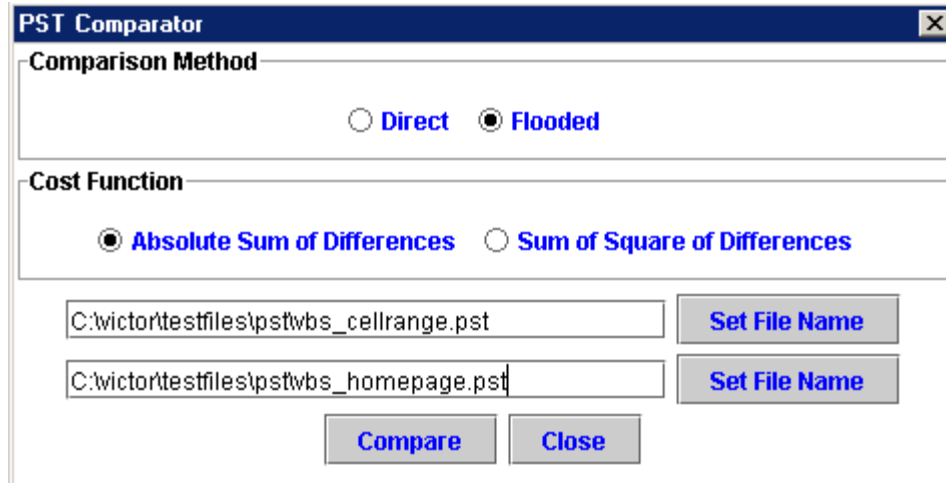


Figure 2.3 – The PST Comparator Dialog Box.

as arguments. The result of the comparison is a double precision number representing the distance between the two trees, where a distance of zero indicates that the trees are identical (see Fig. 2.4).

Once the comparison is finished and the result displayed, the user has two options: either repeat the process (by possibly selecting different trees and comparison settings), or exit the comparator by selecting the *Close* button.

2.3.4 PST Matcher

The PST Matcher is a tool for matching sequences of symbols against a PST, exploring the predictability enabled by such structure. The PST Matcher takes two

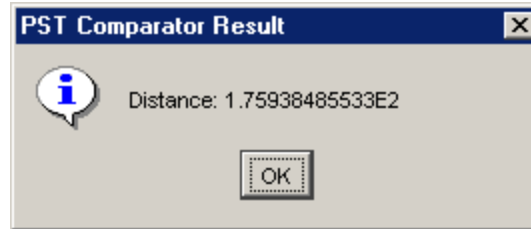


Figure 2.4 – The result of a PST comparison.

inputs: a sample file consisting of a sequence of symbols, and a *.pst* file describing the PST against which the sample is to be matched. The output is an *.mtp* file containing information on the likelihood of the sample by the chosen PST. This file is required by the PST Debugger described in section 2.3.6; this means that if the user intends to use the debugger at some point, the PST Matcher must be executed first.

2.3.5 PST Merger

In malicious mobile code fingerprinting it is often useful to identify behaviors that are common (i.e., behaviors that occur frequently) among certain sets of programs as an attempt to define normal behavior for them. In the context of Windows software, the sequences of system calls these programs perform constitute the fingerprints necessary for the stochastic modeling that PSTs provide, as discussed previously. However, much more insight on normal behavior can be gained if the PSTs for several applications are merged, since sequences that occur more fre-

quently will be easily identified by nodes (and their associated probabilities) that were present in most of the trees.

The PST Merger accomplishes such task by merging two trees at a time. It takes two *.pst* files as input, performs the merging, and writes the resulting tree to a new *.pst* file. This newly created PST keeps a record of the origin of its nodes (i.e., which PST they came from), so that the PST Viewer (described next) is able to highlight specific groups of nodes, thus helping the user to quickly gain a visual insight on a given PST's merging information.

2.3.6 PST Viewer

The PST viewer offers an interactive graphical representation of a PST. It takes as input a *.pst* file containing a binary description of a PST, decodes it into ASCII format, and runs an algorithm to generate a layout for the tree. Once the graph layout algorithm is finished generating coordinates for the nodes and edges in the tree, the PST Viewer displays a window containing three panels, labeled *PST Information*, *PST Display*, and *PST Node Information*, which are described next.

The *PST Information* panel displays general information about the PST currently loaded, such as the number of nodes in the tree, the number of symbols used (i.e., the size of the alphabet Σ), the maximum length for a node's label (in number of symbols), and the tree type ("source" or "merged"). A source tree is a

tree constructed directly from a *.symb* file generated by the Symbol Mapper, whereas a merged tree originates from the merging of two PSTs.

The *PST Display* panel displays a PST as a layered graph. A layered graph is a connected graph where "layers" $L_0 \dots L_k$ partition the vertices, and each edge connects only vertices in successive layers[19]. The graph layout techniques used in SPARTA are discussed in more detail in the next chapter. Nodes are represented by labeled cyan ellipses and are connected by straight lines. A node's label consists of strings of integers separated by underscores, where each integer represents a unique Windows registry entry as specified by the Symbol Mapper aforementioned. A node's bounding box's width is determined according to the length of its label, and if this width exceeds the threshold value `MAX_WIDTH`, the label is truncated to keep a node from spanning the entire screen's width, thus allowing more nodes to be viewed at one time. When a node is selected with the mouse, a Java event listener captures the mouse event and invokes a method that changes the node's color to yellow in order to indicate that it has been selected. This method also updates the *PST Node Information* panel's display to show information pertaining to the node just selected.

The *PST Node Information* panel gives information on a selected node. When no nodes are selected, this panel simply displays the text "<No nodes selected>". Otherwise, it displays four buttons and the selected node's full label (remember

that a node's label may get truncated in the *PST Display* panel when it is too long). Here is the functionality of each of the four buttons:

- **Frequency Counts:** Launches a small window showing the next-symbol frequency counts associated with the selected node.
- **Probabilities:** Launches a small window showing the next-symbol conditional probabilities associated with the node. These probabilities are computed by solving the fractions representing the frequency counts.
- **Create Subtree:** Creates a replica of the *PST Viewer* window displaying a subtree of the original tree, rooted on the selected node. This is intended to give the user the benefit of abstraction by displaying only a small part of a large tree.
- **View System Calls:** Launches a window showing all the registry entries mapped to the symbols in the node's label, starting from the rightmost symbol. This button looks for the *.map* file created by the Symbol Mapper for the correct translation from symbol to registry entry.

The PST Viewer also has a menu bar containing two menus, named *Display Options*, and *Tools*.

The *Display Options* menu contains four features intended to enhance the display of a PST. The first one is a checkbox allowing the user to toggle the *PST Display* panel between two modes: “normal” (the default) and “skeleton”. The “skeleton” mode redraws the PST in a more compact format by shrinking its nodes to small, unlabeled circles, and drawing them closer to each other (see Fig. 2.5). This provides useful abstraction when the interest is on the tree’s shape rather than on its particular nodes, giving the user a better perception of the PST’s density. The second feature in this menu gives the user the option to highlight certain nodes based on their merge information. For instance, if the PST is a merged tree and the user selects the option “Highlight Merged Nodes From PST 1”, all the nodes that came from PST 2 or were present in both trees will be faded out, becoming a blurred cyan ellipse (see Fig. 2.6). The third feature allows the user to save either the entire PST or its visible part as a JPEG image file (see Fig. 2.7). Finally, the fourth feature is a button that displays a list of all the registry calls used in the loaded PST, and their respective integer mapping.

The *Tools* menu consists of three options. The first one is a node locator, which allows the user to search for a particular node in the PST by prompting for a label to be matched against the tree. The second option is the PST Debugger, which prompts the user for an *.mtp* file to be matched against the tree. An *.mtp file*, which is generated by the PST Matcher tool aforementioned, consists of strings of integers separated by underscores and the cumulative probabilities

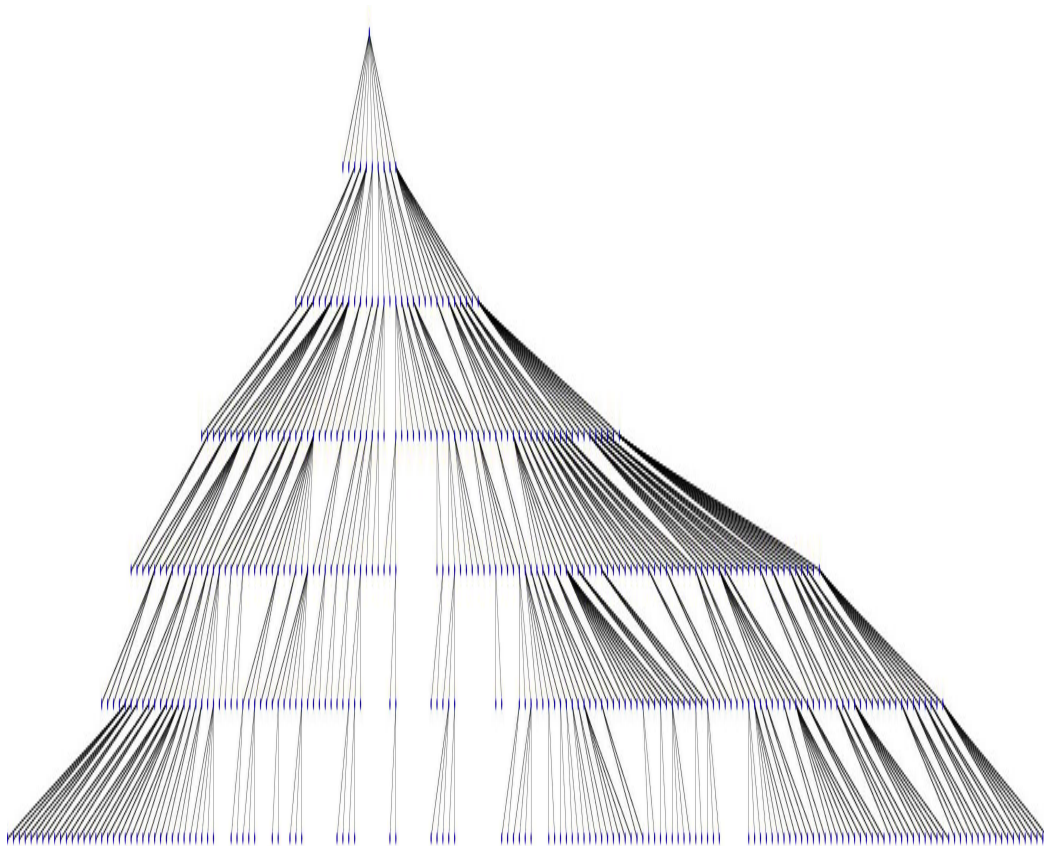


Figure 2.5 – A PST in “skeleton mode”.

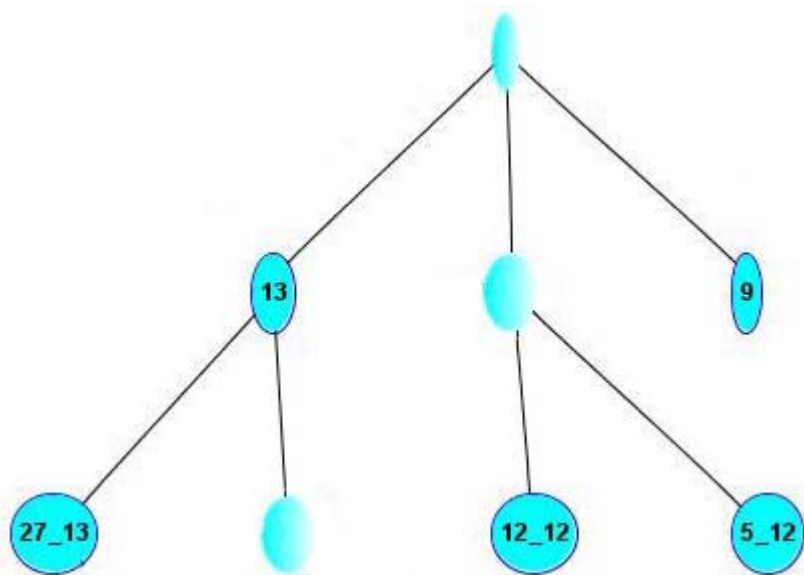


Figure 2.6 – A PST with merged nodes.

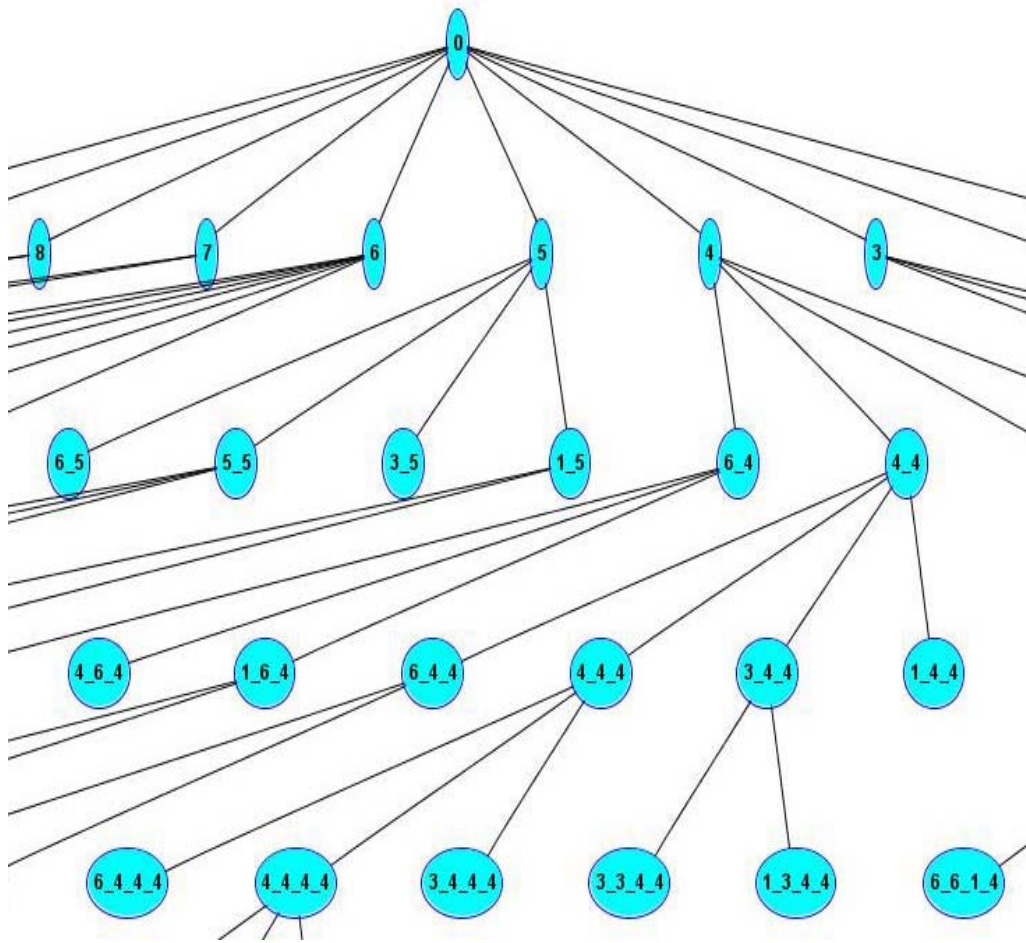


Figure 2.7 – A snapshot of the visible part of a PST

associated with them. Once this file is loaded, a debugger monitor is launched (see Fig. 2.8), presenting the user with options for stepping back and forth along the nodes listed in the *.mtp* file, by centering the *PST Display* panel's viewport¹ around them. The debugger monitor also allows the selection of breakpoints, so that the debugger will only step along nodes containing selected symbols. If the user chooses to select breakpoints, a window displaying the contents of the *.map* file created by the Symbol Mapper containing registry entries and their corresponding symbols along with checkboxes is displayed (see Fig. 2.9), allowing the user to specify registry entries of interest in that particular debugging session. The third feature available in the *Tools* menu allows the user to invoke the PST Matcher in a special mode which partitions the sample in equal-length groups of sample points, and plots these groups (also known as “windows”) and their cumulative probabilities of match in a XY chart (see Fig. 2.10). Partitioning a large sample in relatively small windows keeps its cumulative probability of match from getting too low, thus yielding more meaningful numbers.

¹ A *viewport* is the window displaying the visible part of a panel when it is larger than its bounding frame. The viewport changes upon scrolling, revealing previously hidden (and hiding previously visible) areas of the panel.

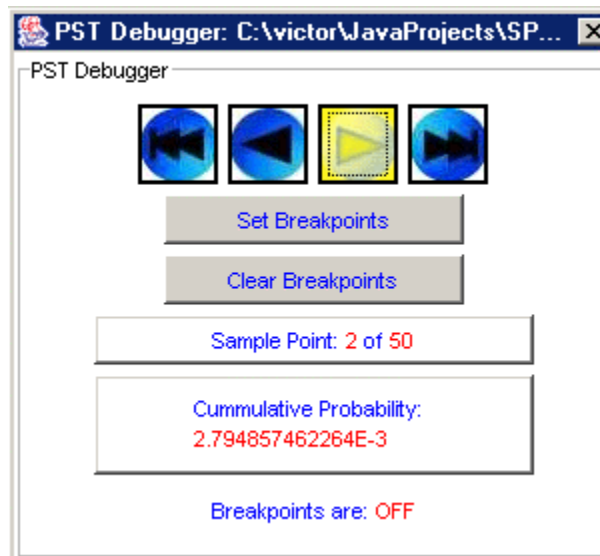


Figure 2.8 – The PST Debugger’s monitor.



Figure 2.9 – The PST Debugger’s breakpoint table

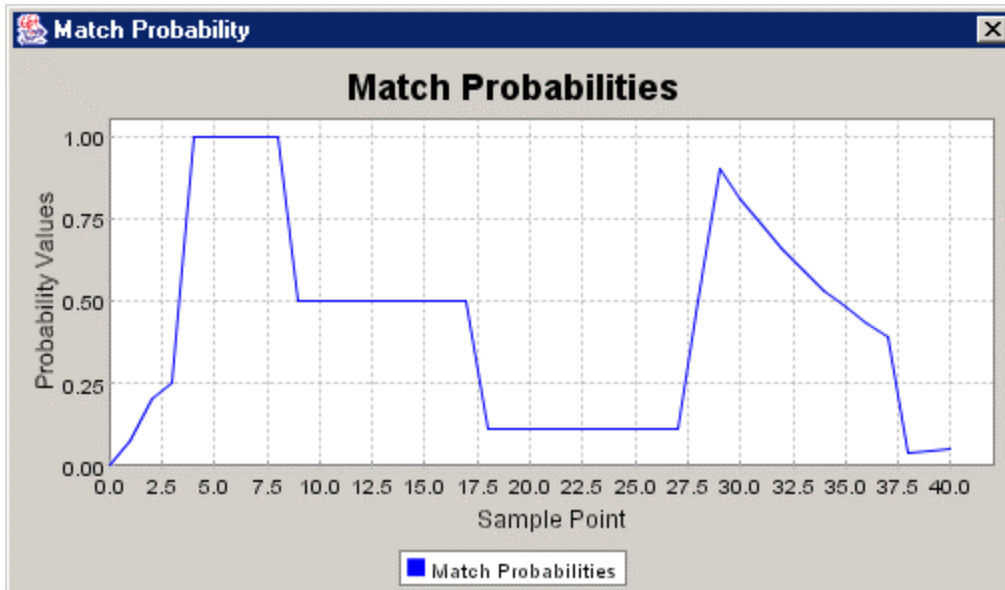


Figure 2.10 – A plot for the cumulative probabilities of match for a sample consisting of 50 symbols partitioned in windows of size of 10.

2.4 Probabilistic Suffix Automata

2.4.1 Introduction

The Probabilistic Suffix Automaton (PSA) represents a subclass of the more general Probabilistic Finite Automaton (PFA) where each state is labeled by a string over an alphabet Σ , and the transition function between any two states is defined in accordance to these string labels, so that a walk on the underlying graph of the automaton, related to a given sequence, always ends in a state labeled by a suffix of the sequence. The lengths of the states' labels are bounded by some upper

bound L , but different states may have labels of different length, indicating varying memory length. When a PSA generates a sequence, the probability distribution on the next symbol generated is completely defined given the previously generated subsequence of length no greater than L . Hence, the probability distributions these automata generate can be equivalently generated by Markov chains of order L , but in a possibly less succinct fashion [9].

It is proven that distributions generated by a PSA can equivalently be generated by a PST[9], and each structure has its own advantages. For example, PSAs generally generate distributions more efficiently and introduce the notion of *stationary distribution* in their states, whereas PSTs can often be represented more succinctly than their equivalent PSAs.

The subsections that follow discuss in detail techniques for building, visualizing, and studying a PSA (see Fig. 2.11), using the tools from the *PSA Actions* menu in the SPARTA suite.

2.4.2 PSA Builder

Exploring the fact that it is possible to characterize the class of PSTs from which PSAs can be derived [9], the PSA Builder tools constructs PSAs from PSTs by reading *.pst* files as input, and generating *.psa* files, which can later be viewed by the PSA Viewer (described in section 2.4.6).

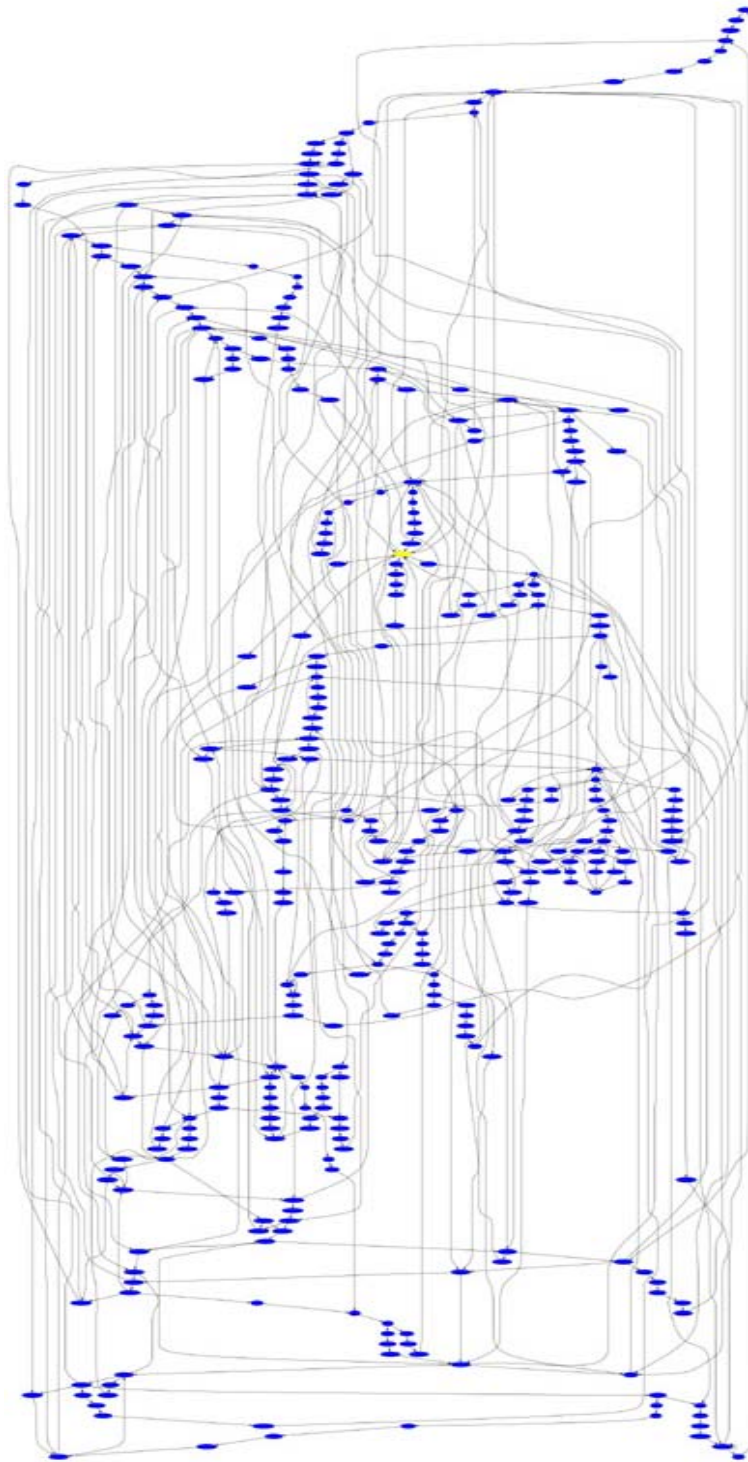


Figure 2.11 – A PSA in “Skeleton Mode”.

Along with the *.psa* file, which contains a description of the nodes and edges of the PSA in binary format, the Builder also creates *.p*, *.m*, and *.w* binary files containing the Π (stationary distribution), Mean, and Variance matrices for the PSA.

After creating the PSA, the builder prompts the user for instantaneous visualization of the automaton.

2.4.3 PSA Condenser

The PSA Condenser is a tool for converting PSA description files to a more compact format, where sequences of nodes connected by single arrows of probability 1 are condensed inside *supernodes*. The PSA Condenser’s only input is a “regular” (non-condensed) *.psa* file; the user is allowed to specify the name and destination of the condensed version of the file.

Combined with the PSA Viewer, the PSA condenser allows SPARTA to generate layouts that are quicker to load and more pleasing to look at, since uninteresting information in a PSA is likely to be hidden in the condensed form. It is important to note that no information is lost in the condensing process: the condensed *.psa* file preserves all the node information present in the regular *.psa* file by making internal nodes accessible from a scrollable popup list that is activated every time a supernode is selected (see Fig. 2.12).

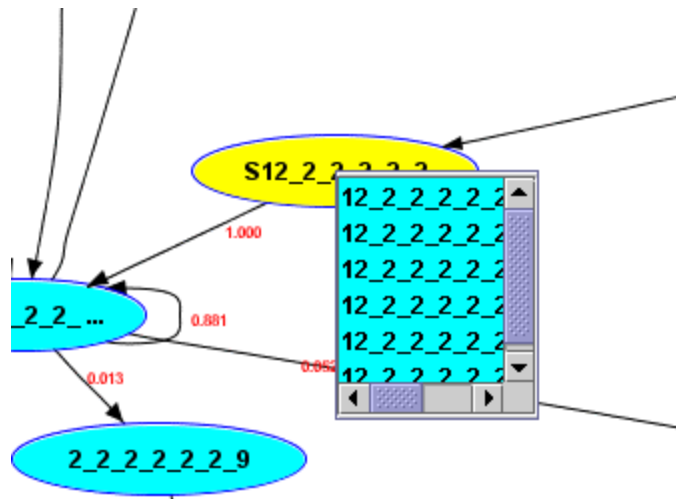


Figure 2.12 – A super node in a PSA. When selected, it displays a scrollable popup list of internal nodes.

2.4.4 PSA Matcher

The PSA Matcher is a tool for matching samples against a PSA, much in the same way as the PST Matcher, in the sense that it also reads as input two files: a sample file consisting of a sequence of symbols, and a *.psa* file containing the description of the PSA against which the sample is to be matched. However, with the PSA Matcher the user has the option of using a *.scn* file generated by the PSA Scanner (described below) with information about trigger states in the matching process, as well as establishing Markovian or Extended Distribution² thresholds (or both), so that the sample is matched against a PSA until either one of these is reached

² Extended distributions are essentially additional information that can be retrieved from the execution log of an application in order to provide a deeper context for a given registry call. Without this context, it may not be possible to identify a registry call as malicious or benign.

(see Fig. 2.13). The results of the matching are placed in an *.anp* file, which is required by the PSA Debugger described in the subsection that follows.

2.4.5 PSA Scanner

Crucial to the success of using a PSA as a model for normal behavior profiling and malicious mobile code fingerprinting is the ability to identify “trigger” states in the automaton, i.e., states that, for some specified reason (high or low frequency, unusual system call mapping etc) should activate or deactivate the fingerprinting mechanism. The PSA Scanner tries to accomplish this goal by searching through a given PSA for these trigger states within user-specified boundaries for the relative frequency (also known as the Π value) and mean values (see Fig. 2.14). The output is a plain text *.scn* file containing pairs of starting and ending trigger states. This output file is then used by the PSA Matcher aforementioned.

2.4.6 PSA Viewer

The PSA Viewer provides a graphical visualization of a PSA. It takes as input a *.psa* file containing a binary description of a PSA, translates it into ASCII format, and passes it to the graph layout program used for PSAs. Before displaying the PSA, the Viewer asks the user whether transient nodes should be displayed: visualizations of PSAs that hide transient nodes result in smaller graphs and offer

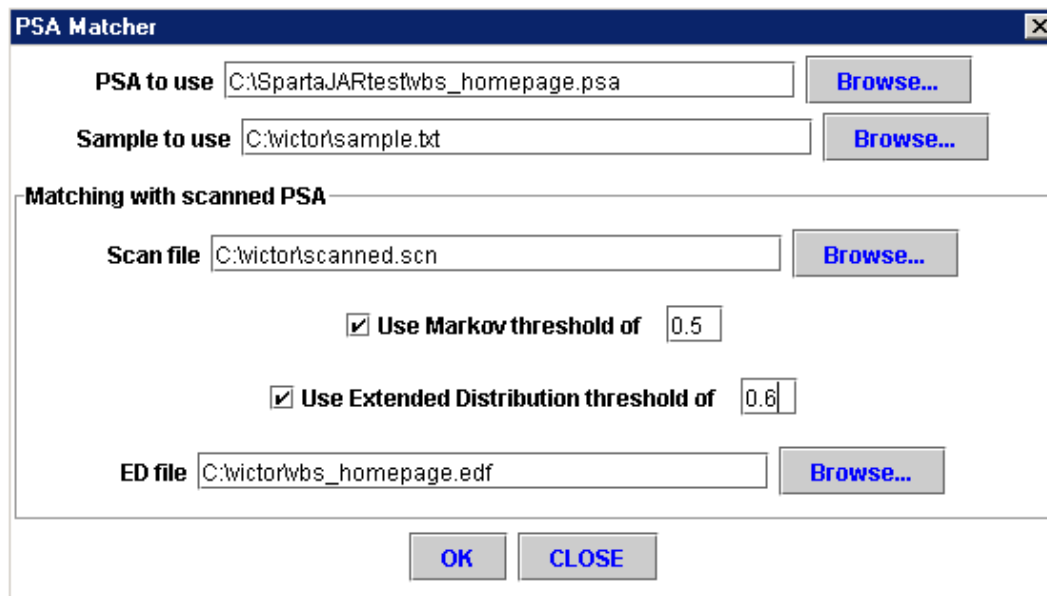


Figure 2.13 – The PSA Matcher dialog box.

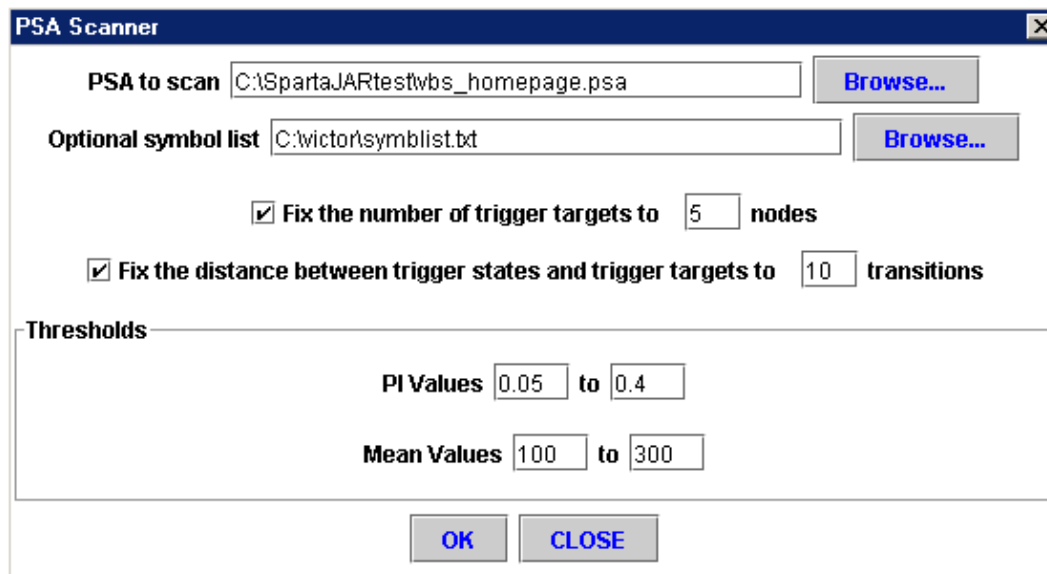


Figure 2.14 – The PSA Scanner dialog box.

greater abstraction, but may cause the PSA Debugger (described ahead) to work incorrectly.

Although a little more sophisticated, the panels in the PSA Viewer are similar in content and purpose as those from the PST Viewer. For instance, the *PSA Display* panel displays a PSA as a layered graph, much in the same way as the *PST Display* panel, except that PSA Edges are represented by arrows, since automata are directed graphs. Also, in an automaton edges are likely to span several layers, thus requiring a more complex routing mechanism and the use of spline segments to connect their several pieces.

The *PSA Node Information* panel contains buttons for the visualization of frequency counts, probabilities, system calls, mean values, and variance values. The first three buttons are analogous to the ones in the *PST Node Information* panel, whereas the latter two load the *.m* and *.w* binary files created by the PSA Builder and display the values associated with the selected node. This panel also displays the relative frequency of the selected node, as well as the node's type (Recurrent or Transient).

The PSA Viewer, like the PST Viewer, also has a menu bar presenting display and debugging options. The *Display Options* menu, in addition to the skeleton mode, snapshot saving, and system call mapping display options, introduces two new options for edge rendering. The first one allows the user to render edges invisible according to the probabilities associated with them. For instance,

the user might not be interested in seeing connections between nodes whose edges have a probability lower than or equal to 0.1; thus, he can set all edges with probabilities in the range 0 to 0.1 to be rendered invisible. The second option allows the user to color-code edges in the automaton where different probability ranges prevail. For example, the user might draw all edges between 0 and 0.3 in green, all edges between 0.3 and 0.6 in red, all edges between 0.6 and 0.85 in purple, and the remaining edges in black. Used with judiciousness, this rendering option can be of great aid in identifying patterns among PSAs; See Figures 2.15 to 2.18 for an example.

The *Tools* menu has similar features to its counterpart in the PST Viewer, except that the PSA Debugger, as opposed to the PST Debugger, flags nodes that were not in the original sample data provided to the PSA Matcher, but were found by backtracking in the sample until a match was found. Those nodes are highlighted in red (see Fig. 2.19).

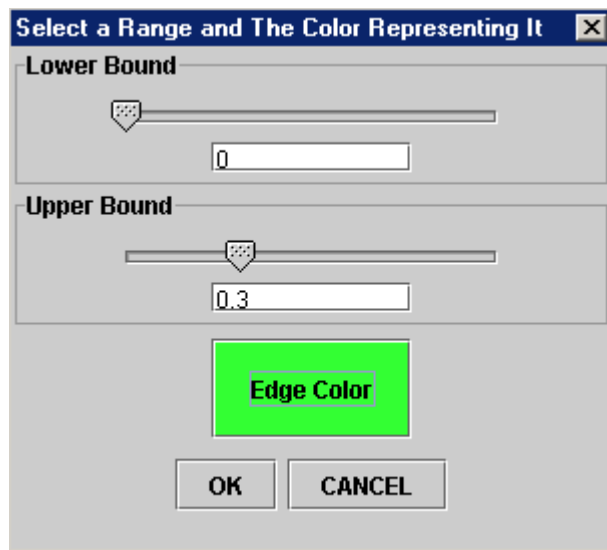


Figure 2.15— The dialog box for probability range color assignment.

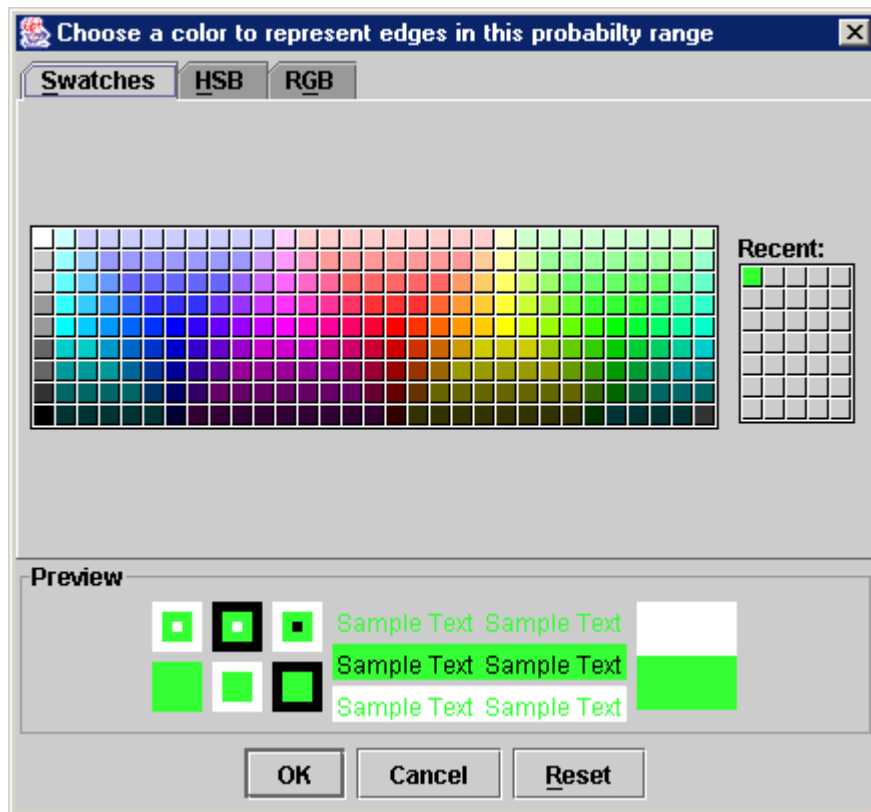


Figure 2.16 – A chooser dialog for edge colors.

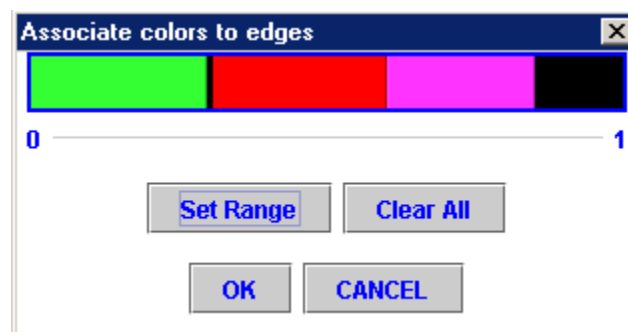


Figure 2.17 – A colored horizontal bar summarizes the probability ranges and their respective color patterns, as established by the user.

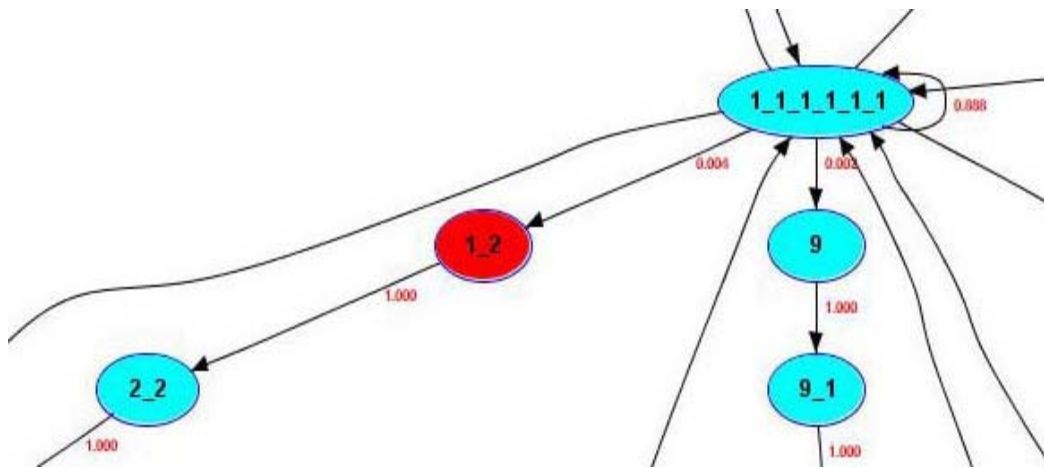


Figure 2.19 – A “backtracked” Node in the PSA Debugger.

Chapter 3

Graph Layout Techniques in SPARTA

3.1 Introduction

Over the past couple of decades, there have been tremendous advances in graph layout techniques. Although the work is still far from being completed (mostly due to the NP-completeness associated with certain steps of current graph layout algorithms), researchers have managed to establish paradigms that are proven to be essential for the “successful” drawing of a graph. In this context, success is measured by the degree to which a graph manages to represent a certain model and the dependencies/relationships among its variables with accuracy and intuitiveness. For example, PERT diagrams³ and subroutine-call graphs are best represented by the *hierarchical approach*, whereas data flow diagrams are usually represented by the orthogonal drawings in the *topology-shape-metrics* approach. The reader is referred to [12] for further details on these and other popular approaches.

³ Standing for *Program Evaluation Review Technique*, a PERT diagram is a project management tool used to schedule, organize, and coordinate tasks within a project, and was first used by the U.S. Navy in the 1950s to manage the Polaris submarine missile program. It consists of a graph where the nodes represent events, or milestones in the project, and the oriented edges represent the tasks in the project [25].

For the purposes of the SPARTA suite, the two most important aspects of graph drawing are:

- ***Aesthetics***: The aesthetic goal in graph drawing is to minimize edge crossings, edge bends (so that edges look as straight as possible), total edge length, and total area encompassed by the graph, while maximizing the display of symmetries. This last criterion is of especial importance in SPARTA, since the ultimate goal in the visualization of PSTs and PSAs is to reveal patterns.
- ***Computational Efficiency***: Since SPARTA is intended to be a highly interactive tool, it must provide real-time response to user events. Therefore, even the largest drawings must be laid out in a timely fashion.

It is important to notice that the aesthetic criteria mentioned above often conflict with each other, making it next to algorithmically infeasible to enforce them all at the same time. Tradeoffs are thus unavoidable, and the final layout must rely on heuristics to obtain the best compromise among all the aesthetic criteria adopted and still generate an acceptable visualization of a graph.

SPARTA relies on two independent algorithms to generate layouts for its graphs. PSTs are constructed using a Java port of *Graphplace*[15], originally developed by Jos van Eijndhoven at the Eindhoven University of Technology, whereas PSAs are constructed with the sophisticated *Dot* algorithm, developed in

C by Gansner, Koutsofios, North, and Vo at the AT&T Bell Labs[10]. The next section describes the basic algorithm common to both tools, followed by an analysis of the strengths and weaknesses of each tool.

3.2 The Basic Algorithm

Since PSTs and PSAs are basically modeling dependency relationships, the *hierarchical approach* is adopted in SPARTA. The standard algorithm based on this approach takes as its input a graph description (formally, a set V of vertices and a set E of edges), and generates coordinates for its vertices and edges in four steps:

1) **Cycle Removal:** This is an optional step that takes place whenever the graph contains cycles. Since it is often aesthetically pleasing to generate graphs where most edges flow in the same direction (e.g., from top to bottom), this step transforms a cyclic graph into an acyclic one by reversing some of its edges. The graph is then laid out according to the next three steps, and all reversed edges are restored to their original directions before the algorithm returns.

2) **Layering:** This step transforms the acyclic input graph into a *proper layered digraph*, where its vertices are assigned to layers $L_1, L_2, \dots, L_n$ such that the bounding boxes of vertices sitting on the same layer have the same top y-coordinate, and edges never span more than one layer. If the endpoints of an edge are more than one layer apart, then *dummy vertices* (also known as *virtual verti-*

ces) are inserted across the layers to enforce the one-layer-span rule. Some of the aesthetic concerns involved in this step are the final width and height of the graph (measured by the size of its bounding box), and the number of dummy vertices created, since it can be quadratic if there are $O(n)$ edges spanning $O(n)$ layers.

3) ***Crossing Reduction:*** As the name suggests, this step receives as an input a layered digraph, and reorders the vertices on each layer in such a way that the number of edges crossing is minimum. The problem of reducing edge crossings is in fact the combinatorial one of choosing appropriate vertex orderings for each layer, and unfortunately is NP-complete [18]. This problem can be dealt with in a variety of ways, all of them applying some sort of heuristics; the most common approaches are the *Barycenter* and *Median* methods [12], since they run in linear time and are capable of generating planar layouts (i.e., layouts where no two edges cross) whenever one exists for a given input graph.

4) ***Horizontal Coordinate Assignment:*** This step assigns x-coordinates to the vertices in each layer, respecting the ordering computed in the crossing reduction step. Further aesthetic refinement in the drawing can be achieved in this step by vertically aligning dummy vertices, since this reduces the number of bends in long edges. Although conceptually simple, this step relies on quadratic programming techniques [8] that require a great deal of computational effort.

3.3 Graphplace Vs. Dot

3.3.1 Graphplace

Because of their planar representation, trees can usually be laid out much more easily than a generic graph. For this reason, Graphplace is used to render PSTs in SPARTA. This algorithm offers a simple, yet effective implementation of the graph layout algorithm described in section 3.2. It bestows a very fast linear-time heuristic for node placement, traversing the graph in a Top-Down (North-South), depth-first search order along the edges, which are just assumed to be directed (this has no implication in the layout of undirected graphs, which is the case of PSTs). Furthermore, the Java port of Graphplace used in SPARTA completely ignores the cycle removal step, since trees are by definition acyclic undirected graphs. The overall computational complexity of Graphplace is $O(n \log n)$, where n is the number of vertices, which allows fast placement of even very large graphs. The limitations in the placement provided by Graphplace rest in the following facts:

- Nodes are assumed to be represented as dots (i.e., they have no area). This can cause nodes to overlap if their areas cover the vertical / horizontal space that separates them. In SPARTA, where PST nodes are represented by ellipses, this problem was fixed by speci-

fying maximum values for the widths and heights of the nodes' bounding boxes (which explains the maximum label length criterion described in chapter 2) and making the horizontal and vertical distances between any two nodes greater than those values;

- The layout technique implicitly assumes that edges are drawn as *polylines* containing the dummy nodes created in the layering step as vertices in a polygonal chain. This is not noticeable in PSTs, since no dummy nodes are created and hence edges are always drawn as straight lines. However, the display of a more generic graph such as an automaton would be impaired by a polyline approach, as it would be more desirable and intuitive to draw the edges as spline segments.

These limitations are next to harmless in tree drawing, and in fact help the Graphplace algorithm run faster. They do, however, show why Graphplace would not be as effective in the layout of more general graphs. For one thing, the assumption that nodes are represented as points could not be handled so easily, since overlaps might occur not only between two ordinary nodes, but also between nodes and dummy nodes, which would result in edges running over nodes other than its endpoints or control points. If on the one hand dimension constraints could be applied to the nodes' bounding boxes to avoid node-node overlap, applying the same con-

straints to dummy nodes would be likely to result in unnecessarily large layouts that would fail to wisely explore the spaces between nodes in the routing of edges. Furthermore, the simple heuristics used in the cross reduction and x-coordinate assignment steps of Graphplace have proven to be less intuitive and effective than desired, generating layouts that were hard to assimilate because of the excessive number of edge crossings and the sharp bending of long edges.

3.3.2 Dot

The need for a more powerful layout algorithm for drawing PSAs led to the use of Dot. Its algorithm offers improvements on all the major issues with Graphplace aforementioned:

- Node dimensions are considered when generating coordinates for their bounding boxes. These dimensions can be custom-specified, and widths can even be automatically calculated based on the label lengths of the nodes;
- A more complex version of the Median method is used for crossing reduction, resulting in “cleaner” drawings and more straight edges. This is very useful when displaying large PSAs with numerous nodes and edges;

- A powerful spline routing mechanism, which transforms the dummy points of the layering step into B-spline control points is used. This results in very smooth splines, reducing the “spaghetti effect” caused by unnecessary bends in an edge that was so prominent in the PSA layouts generated by Graphplace. The spline routing mechanism also supports self-edges (or “loops”) in the layout, which are quite common in automata.

Dot is also innovative in that it relies on a network simplex algorithm (NSA) during the steps that require heuristics. The method is a variation of the popular Linear Programming algorithm known as *Simplex*, which basically tries to find an optimal solution to a problem by iteratively generating intermediate (and increasingly better) solutions until an optimality criterion is met. The network version of the algorithm tries to make the number of iterations smaller by approaching the problem in a geometrical, graphical context, rather than as a numerical one. The NSA is used in two occasions in the layout process. First, to solve a linear programming problem (which can be translated into an integer programming problem in polynomial time) of generating a "ranking" (i.e., a certain vertical ordering of nodes analogous to the layering step of the basic algorithm aforementioned) of nodes that results in the shortest edges (whose lengths are associated with “weights” which have to be greater than a certain threshold, usually

one). The NSA is used again in the assignment of x-coordinates by solving the problem of finding an arrangement of nodes along the x-axis where edges between nodes in adjacent ranks remain as close to vertical as possible. In other words, the NSA tries to assign close x values to the endpoints of a given edge. These two instances of the algorithm (especially the second one) are used because they are easy to program, but as a drawback they can be rather costly for medium to large graphs, since the size of the simplex matrix can grow from VE to $2VE+E^2$ entries (where V is the number of vertices, and E the number of edges).

The layout speed problem in Dot caused by its NSA can be a rather disheartening one in an interactive application such as SPARTA. This problem is dealt with in the PSA Viewer in the following way: graph layout information (i.e., the ASCII output of Dot) is stored in a *layout* file, so that whenever a *.psa* file is loaded, the viewer will first look for this file (which will be homonymous in prefix to the *.psa file*, with either the *.sgraph* or *.fgraph* extensions, standing for "simple" graphs without transient nodes, or "full" graphs, respectively). If the file is present and has the same timestamp of its corresponding *.psa* file, it is used as the graph description and the execution of Dot is bypassed (which makes the rendering almost instantaneous). Otherwise Dot is invoked to generate the graph layout file (which will of course be marked with the same timestamp of its corresponding *.psa* file). This is not a bad solution, since users are more likely to work

with already created PSAs, rather than constantly creating new ones and then disposing of them.

3.3.3 Conclusion

In general terms, Dot is undeniably more powerful than Graphplace, insofar as its more sophisticated heuristics result in more intuitive drawings that manage to find the best compromise among all the aesthetic criteria adopted in the *hierarchical approach*. It is certainly indispensable for laying out complex digraphs such as PSAs. However, Graphplace has the advantage of being used as a Java package running in $O(n \log n)$ time, thus being much faster than Dot (which is executed as a native program). Besides, Graphplace often generates good drawings of simpler structures such as trees, which makes it the algorithm of choice for rendering PSTs (see Fig. 3.1).

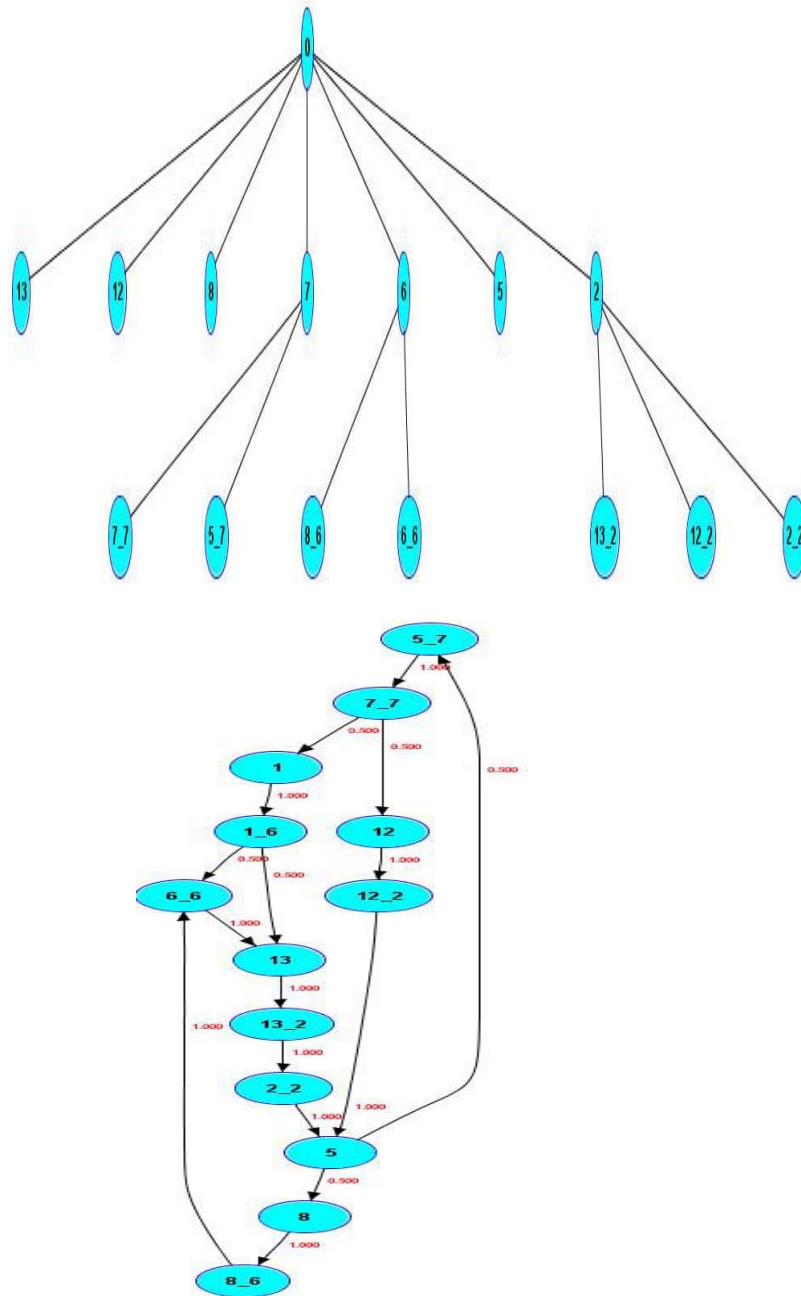


Figure 3.1 – A PST laid out with Graphplace and a PSA laid out with Dot.

Chapter 4

Design Concerns and Important Algorithms

4.1 Java as the Development Platform

From the very start, the SPARTA suite was intended to be an easily maintainable and expandable graphical interface for the several applications it includes. Ergo, the development platform of choice was Java, which is a powerful and well-documented object-oriented programming language. In particular, the following Java features proved to be of utmost usefulness during the development of SPARTA:

- **Documentation:** The Java language is very well documented and has a very informative on-line API. Each class in the JDK (Java Development Kit) is fully described, with examples and references to the Java tutorial (available at <http://java.sun.com/docs/books/tutorial/>). Furthermore, there are dozens of Java-related discussion groups and chat channels on the web, which have proved very helpful in dealing with the common-yet-not-trivial GUI development aspects of a sizeable application such as SPARTA;

- **Enforced Exception Handling:** Java requires that a method either catch those exceptions checked by the runtime system or specify that it can throw that type of exception. This forces the programmer to put more thought into the code, and provides an effective way to quickly locate the source of a bug and identify its nature as the code gets bigger and increasingly complex. For instance, when SPARTA invokes the PST Viewer (which is a binary executable), it does so in the following fashion:

```
Try{
    Runtime application = Runtime.getRuntime();
    Process proc = application.exec(input);
    ...
}
catch(Exception e){
    e.printStackTrace();
    return;
}
```

This code tries to execute the command denoted by the string *input* (in this case, the PST Viewer path and an argument for it), and returns an error message when an exception is caught. The nature of the exception is denoted by the variable *e* (in this scenario, *e* is likely to report an I/O exception). Note that if the *try* and *catch* statements were not present, the java compiler would yield an error message alerting the user to handle any possible exceptions.

- **Portability:** Java is cross-platform, which means that source code written in that language is fully portable to any platform where the JVM (Java

Virtual Machine) is installed. Since malicious mobile code fingerprinting is not a Windows-only need, the SPARTA interface can be easily re-used in other platforms such as UNIX and Mac (though the actual binaries SPARTA uses will have to be re-compiled for those platforms).

- ***Powerful String Manipulation:*** A significant portion of the code in SPARTA deals with the parsing of textual data, such as user input or descriptions of trees and automata. Text parsing, which usually is a painstaking and error-prone process in languages such as C, is almost trivial in Java. For instance, all text parsing was concisely and conveniently performed through the methods from the *String* and *StringTokenizer* classes. Furthermore, the method `substring(int i, int j)`, which extracts a substring from a string object, made it very practical to manipulate file names where the extension needed to be removed or replaced.

4.2 The Graph Rendering Scheme

Most models based on real data correspond to fairly large and complex graphical representations (as is often the case with PSTs and PSAs), which make it infeasible to fit the entire graph on the screen. One possible remedy is to scale the layout according to the size of the graph, but this can be impractical because it can cause nodes to be rendered too small, making it difficult to read their labels. Another

plausible solution would be to implement zooming so that particular regions of a graph could be observed in more detail, but this would still be ineffective because of the possible slowdown caused by re-rendering the graph every time a zoom action was performed, which would be somewhat of a nuisance for the user. A better alternative would be to implement scrolling so that a graph can be rendered in its full size and according to the aesthetic criteria outlined in chapter 3. Taking these visualization aspects into consideration, SPARTA’s rendering scheme makes use of scrollable panels, which are as large as required by the graphs. The resulting display is a compromise between seeing a good portion of the PST/PSA graphs in the viewport at one time and easy visualization of nodes’ labels. Besides, the user can always obtain a more structural visualization (i.e., a visualization that emphasizes shape over detail) of the graphs by switching them to “skeleton” mode.

In order to define the window of visible (and therefore renderable) nodes in the panel given the position of the viewport at one time, it was necessary to re-group nodes in a data structure capable of updating itself in an efficient fashion upon scrolling events. This data structure was called *VertexWindow*, and was implemented over a graph in three stages:

- **Stage 1:** A naïve implementation, where the entire graph was re-drawn whenever a scroll action was performed; this was extremely

inefficient, and for medium-to-large graphs (200 or more nodes) scrolling was painfully slow, taking several seconds for every pixel of movement of the viewport;

- **Stage 2:** The first real implementation of the *VertexWindow* structure. An original internal structure called *X-Y Vectors* was bestowed and brought forth substantial efficiency to scrolling events, inasmuch as only the visible parts of the graph were being redrawn. However, scrolling still was not real-time;
- **Stage 3:** A new and much more elegant implementation of *VertexWindow* using kd-trees [17] was implemented. The real-time scrolling enabled by this structure provided an easy way to hover around the graph.

In the subsections that follow, the *X-Y Vectors* implementation of the *VertexWindow* structure is described in detail, and then compared with the kd-tree approach.

4.2.1 X-Y Vectors

This structure can basically be described by two vectors, namely *vArray_X*, and *vArray_Y*, and four integer variables called *head_x*, *tail_x*, *head_y*, and *tail_y*.

The first vector groups vertices in sorted order by x-coordinates, and the latter one

groups vertices in sorted order by y-coordinates; the integer variables are indices in the *vArray* vectors and indicate the leftmost, rightmost, topmost, and bottommost nodes respectively.

Once a graph is loaded into memory, we initialize an instance of *VertexWindow* by setting up its vectors and the vectors' indices. These indices are computed with respect to the current position of the viewport of the panel where the graph is to be rendered. The left, top, width and height fields of the viewport are used to specify the *VertexWindow* indices. Then, all vertices that are in the ranges $[head_x, tail_x]$ and $[head_y, tail_y]$ at the same time are visible and therefore drawn in the panel. After an instance of *VertexWindow* is initialized, its state may change if one of five possible actions occurs in the viewport: a left scroll, a right scroll, an up scroll, a down scroll, or a resize. These events will cause the viewport coordinates to change, which in turn will require the *VertexWindow* instance to readjust its indices, if necessary. Note that a state change in the viewport will not necessarily cause the *VertexWindow* state to change, since the range of visible nodes might not change after a scroll or a resize event.

Complexity Analysis: Most of the time spent in the initialization is due to the merge sorting of the node vectors, which in the worst case takes $O(n \log n)$ time (in fact, in most cases it will take less than that, since the code explores the nature of the node layering algorithm to generate vectors that already are nearly sorted). The focus of this analysis is thus on the crucial aspect of this algorithm:

the range query, which takes place every time the state of the viewport changes. In the current implementation of the algorithm, range queries take $O(n \log n)$, since every node in the vector delimited by $[head_x, tail_x]$ has to be binary-searched in the vector delimited by $[head_y, tail_y]$. Thus, the efficiency of this algorithm depends heavily on the way the SPARTA graph viewers are implemented: the size of the viewport, the nature of the layering, and the size of the nodes' bounding boxes must ensure that only a few nodes are visible at one time, so that even in the worst case, when all vertices have the same x or y coordinates and the viewport is moved from an empty area to an area with k vertices ($k \ll n$, where k is the maximum number of vertices that can fit in the viewport and n is the total number of vertices), range queries will take $O(k \log n) \ll O(n \log n)$.

4.2.2 X-Y Vectors Vs. Kd-Trees

The literature indicates that among “several exotic structures that support range searching”, k-d trees achieve “respectable running times”[16]. Constructing a two-dimensional kd-tree (also known as a 2d-tree) for SPARTA nodes (where all the items will have arrived before queries are processed) takes $O(n \log n)$ in the worst case, which brings no gain over the *X-Y Vectors* algorithm; in fact, it may perform worse on average, since the *X-Y Vectors* algorithm explores the nearly sorted nature of vectors to achieve linear time in many cases. Range queries in a perfectly balanced 2d-tree (which is the case in SPARTA, since all nodes are

added at construction time, with no further insertions or deletions once the tree is created) take $O(m + \sqrt{n})$, where m is the number of matches. This is clearly faster than $O(k \log n)$, especially if taken into consideration that “for typical n , the difference between $\sqrt{n}$ and $\log n$ is compensated by the smaller constant that is hidden in the Big-Oh notation”[16]. It is thus easy to see that a kd-tree is the data structure of choice for 2-dimensional range querying in SPARTA: it is easy to implement (especially if by recursion) and, given the assumption that it will always remain balanced, offers faster querying times. To further warranty this analysis both of the implementations of *VertexWindow* were tested thoroughly and the empirical results confirmed the expectations: the algorithms had similar performance when only but a few nodes were visible at a time; however, when SPARTA was switched to “skeleton mode” (which makes more nodes visible by reducing their sizes and the distances between them), the 2d-tree implementation offered a much faster scrolling speed, which was the gauge used to measure the range query time.

A final consideration is in order. If in the future SPARTA offers support for insertion and deletion of nodes to its graphs, the 2d-tree implementation of *VertexWindow* might be outperformed by the *X-Y Vectors* algorithm. This is because the merge sort algorithm used by *X-Y Vectors* will always re-sort the *vArrayX* and *vArrayY* vectors in linear time for insertion (since the resulting vector will be nearly sorted), and in constant time for deletion. A 2d tree, on the other

hand, will have to be rebalanced to maintain its $O(m + \sqrt{n})$ range query time, and this is usually accomplished by reconstructing its sub-trees, which is a rather cumbersome process and will certainly take longer than resorting a nearly sorted vector.

4.2.3 Edge Drawing Techniques

The edge drawing process is approached in two different ways in SPARTA, depending on whether the graph is a PST or a PSA. Edges connecting nodes that are separated by at most one layer are relatively straightforward to render using straight lines, whereas graphs where nodes can be two or more layers apart require a more complex approach using piecewise cubic splines. The first approach is applied to the rendering of PST edges, and the latter one is applied to PSA edges. These two approaches are described in detail in the next two subsections.

4.2.3.1 PST Edges

Since nodes in a tree are by construction one layer apart from their parents, straight lines can be used to connect a PST node and its offspring. In SPARTA, this is accomplished by defining the endpoints of an edge to be the center of the parent's bounding box, and the nearest point on the children's elliptical shape. Let

P denote the parent node's center point, and P' its nearest point on the child's ellipse. $P = (x_c, y_c)$ can be easily calculated by the following formulas:

$$x_c = x + w/2$$

$$y_c = y + h/2,$$

where x and y are the left and top coordinates, and w , and h are the width and height of the parent node's bounding box. Obtaining P' is more problematic, since its calculation depends on equations of polynomials of degree 4, for which there is no closed-form solution[8]. A numerical approximation could be attempted, but this could slow down the rendering process significantly. Since geometric accuracy is not absolutely essential in the context of PST rendering (the only hard requirement is that P' lies on the child's ellipse), it is possible to use an altered version of the formula for the nearest point on a circle to generate acceptable results. This formula for $P' = (x_n, y_n)$ is given below:

$$x_n = x'_c + \frac{\frac{w'}{2}(x_c - x'_c)}{\sqrt{(x_c - x'_c)^2 + (y_c - y'_c)^2}} \quad y_n = y'_c + \frac{\frac{h'}{2}(y_c - y'_c)}{\sqrt{(x_c - x'_c)^2 + (y_c - y'_c)^2}}$$

where x'_c , y'_c , w' , and h' are the child node's analogous values to x_c , y_c , w , and h .

4.2.3.2 PSA Edges

PSAs do not preserve the one-layer-distance rule for every parent/child pair, although the layout algorithm described in the previous chapter creates invisible (dummy) nodes in each layer separating a node from its child. The first approach at PSA drawing in SPARTA used these invisible points as vertices for a polyline connecting a node and its child, following the same idea for drawing edges in a PST. This approach was immediately dismissed as ineffective, insofar as the resulting graph's edges had sharp bends, making the PSA hard to read and interpret. Furthermore, edges in automata are usually represented by smooth curves, and the polyline approach is a major departure from that convention.

The current PSA edge-drawing approach makes use of the fact that the invisible nodes generated by the layout algorithm do not have ordinary coordinates, but are actually centered on B-spline control points. Nevertheless, before these points can be used, they must first be converted into Beziér control points, since that is the only geometry supported by Java's *CubicCurve* object [27]. The conversion is straightforward, and uses the Beziér and B-spline characteristic matrices (denoted by $\mathbf{B}_z$ and $\mathbf{B}_s$, respectively) shown below:

$$\mathbf{B}_z = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \mathbf{B}_s = 1/6 * \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}$$

Define $P(i)$ to be a b-Spline control point, with i ranging from 1 to $n-3$, where n is

the number of control points. Then, $\mathbf{P} = \begin{bmatrix} P(i-1) \\ P(i) \\ P(i+1) \\ P(i+2) \end{bmatrix}$ can be converted into $\mathbf{P}'$

in Beziér geometry through the following formula:

$$\mathbf{P}' = \mathbf{B}_z^{-1} \mathbf{B}_s \mathbf{P}$$

Another issue involved in the drawing of PSA edges is the fact that automata are directed graphs, and hence one of the endpoints of an edge must be combined with an arrowhead. Since Java does not provide an arrowhead object, the problem was approached geometrically. Specifically, arrowheads are represented as isosceles triangles whose heights are collinear with the line connecting two points, namely $\mathbf{A}$ (the closest control point to the child's ellipse) and $\mathbf{B}$ (the closest point on the child's ellipse). Let the subscripts $_x$ and $_y$ denote the horizontal and vertical coordinates for a given point, and $\mathbf{R}$ be a constant positive integer

value. Then, the vertices $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$ of the triangle are obtained through the following formulas:

$$\begin{aligned} P_0 &= (B_x - D_x R + D_y \frac{R}{2}, B_y - D_y R - D_x \frac{R}{2}) \\ P_1 &= (B_x, B_y) \\ P_2 &= (B_x - D_x R - D_y \frac{R}{2}, B_y - D_y R + D_x \frac{R}{2}) \end{aligned}$$

where

$$D_x = \frac{B_x - A_x}{\sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}} \quad D_y = \frac{B_y - A_y}{\sqrt{(B_x - A_x)^2 + (B_y - A_y)^2}}$$

Without loss of generality, assume the arrowhead is pointing upwards (otherwise rotate its vertices so that it points upwards). Then $\mathbf{P}_0$ is the left base vertex, $\mathbf{P}_1$ is the top vertex, and $\mathbf{P}_2$ is the right base vertex. The arrowhead is rendered by connecting the points $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$.

Chapter 5

Conclusion and Future Work

This thesis introduced SPARTA, a Java-based graphical user interface for a suite of malicious mobile code detection tools. The primary goal of SPARTA is to explore the intrinsically graphical nature of the PST and PSA models in order to reveal software behavioral patterns that might otherwise go unnoticed in the ASCII output generated by the detection tools. The ultimate goal of SPARTA is to be able to use the properties of these models to identify common behavior among certain groups of Windows applications, so that a profile of what is considered normal or acceptable behavior can be built, and (possibly malicious) deviations from it can be identified.

The near-future plan is that once the normal behavior profiling goal is accomplished, SPARTA will be interfaced with HEAT in order to expand the system-monitoring tool's ability to detect not only known intrusions, but also new malicious signatures, making it more robust and reliable. Long-term plans for SPARTA might include porting the code to a compiled language such as C# so that performance may be gained on a given platform, and the incorporation of a graph layout algorithm for PSAs that will run at least as fast as Graphplace, while still preserving the quality achieved by Dot. This is a reasonable expectation, in-

asmuch as fast graph layout algorithms is the subject of intense research in computer science.

A final consideration is that the techniques for malicious mobile code detection explored by the tools in SPARTA are general enough that they can be applied to platforms other than Windows, which makes the portability of the code for SPARTA a handy feature. As a matter of fact, as long as there is a Java Virtual Machine available for a given platform, SPARTA does not even have to be recompiled, although the binaries on which it relies upon will have to be made available for the target platform.

Bibliography

- [1] B. Vander Zanden and B. Myers. "Demonstrational and Constraint-Based Techniques for Pictorially Specifying Application Objects and Behaviors". ACM Transactions on Computer Human Interaction, vol. 21, no. 4, Dec. 1995, pp. 308-356.

- [2] C. Warrender, S. Forrest, and B. Pearlmutter. "Detecting Intrusions Using System Calls: Alternative Data Models". In IEEE Symposium on Security and Privacy, IEEE-Computer Society, 1999, pp. 133-145.

- [3] D. Anderson, T. F. Lunt, H. S. Javitz, A. Tamaru, and A. Valdes. "Detecting Unusual Program Behavior Using the Statistical Component of the Next-generation Intrusion Detection Expert System (NIDES)". Technical Report SRI-CSL-95-06, SRI Computer Science Laboratory, May 1995.

- [4] D. E. Denning. "An Intrusion Detection Model". In IEEE Transactions on Software Engineering, Los Alamos, CA, IEEE Press, 1987, pp. 222-232.

- [5] D. Farmer, E. H. Spafford. "The COPS Security Checker System". Technical Report CSD-TR-993, Purdue University, Comp. Science, 1996.

- [6] D. Farmer, W. Venema. "Improving the Security of Your Site by Breaking Into It". Online. Available: <http://www.fish.com/security/admin-guide-to-cracking.html>. Accessed Nov. 2, 2002.
- [7] D. M. Geary. Graphic Java 2. Swing, Vol. 2, Palo Alto, CA: Prentice Hall, 1999.
- [8] D. R. Olsen, Jr. Developing User Interfaces. San Francisco, CA: Morgan Kaufmann, 1998.
- [9] D. Ron, Y. Singer, and N. Tishby. "The Power of Amnesia: Learning Probabilistic Automata with Variable Memory Length". Machine Learning, Vol. 25, No. 2-3, 1996, pp. 117-149.
- [10] E.R. Gansner, E. Koutsofios, S.C. North and K.-P. Vo. "A Technique for Drawing Directed Graphs". In IEEE Transactions on Software Engineering, Vol. 19, No. 3, March 1993, pp. 214-230.
- [11] G. Bejerano and G. Yona. "Variations on probabilistic suffix trees: statistical modeling and prediction of protein families". Bioinformatics, Vol. 17, No.1, pp. January 2001, pp. 23-43.

- [12] G.Di Battista, P. Eades, R.Tamassia, and I.G. Tollis. Graph Drawing: Algorithms for the Visualization of Graphs. Upper Saddle River, NJ: Prentice Hall, 1998.
- [13] J. A. Whittaker and A. De Vivanco. “Neutralizing Windows-Based Malicious Mobile Code”. In Proceedings of the 17th ACM Software Applications Conference (ACM-SAC), 2002. Madrid, Spain.
- [14] J. A. Whittaker, R. Menezes, C. Kaner, M. G. Thomason, J. Gregor, G. A. Marin. “Proactive, Selective Behavior Filtering with Data Restoration: A New Technique for Runtime Neutralization of Malicious Mobile Code”. Office of Naval Research grant proposal, 2001.
- [15] J. T. J. van Eijndhoven. Graphplace. Eindhoven University of Technology. Online. Available:
<ftp://ftp.dcs.warwick.ac.uk/people/Martyn.Amos/packages/graphplace/>.
- [16] M.A. Weiss. Data Structures & Algorithm Analysis in C++. New York, NY: Addison-Wesley, 1999.

- [17] M. deBerg, M. vanKreveld, M. Overmars, O. Schwarzkopf.
Computational Geometry. New York, NY: Springer-Verlag, 2000.
- [18] M. R. Garey and D. S. Johnson. Computers and Intractability. New York,
NY: W. H. Freeman and Company, 2002.
- [19] National Institute of Standards and Technology. “Layered Graph”.
Online. Available: <http://www.nist.gov/dads/HTML/layeredgraph.html/>.
Accessed Jul. 6, 2002.
- [20] National University of Singapore. “Sequence Classification Methods”.
Online. Available:
http://xena1.ddns.comp.nus.edu.sg/CBL/doc/Xie_Zhipeng.ppt/. Accessed
Jul. 4, 2002.
- [21] P. H. Winston, S. Narasimhan. On To Java 2. New
York, NY: Addison-Wesley, 2001.
- [22] P. Norton, M. Groh. Peter Norton's Guide to Visual Basic 6.
Indianapolis, IN: Sams, 1998.

- [23] S. Forrest and S. A. Hofmeyr. "Immunology as Information Processing". Design Principles for Immune System & Other Distributed Autonomous Systems. Oxford: Oxford University Press, 2000, pp. 361-387.
- [24] S. Hofmeyr, S. Forrest, and A. Somayaji. "Intrusion detection using sequences of system calls". Journal of Computer Security, vol. 6, 1998, pp. 151-180.
- [25] searchSystemsManagement.com. "PERT Chart". Online. Available: <http://searchsystemsmanagement.techtarget.com/>. Accessed Jul. 15, 2002.
- [26] T. Wu, M. Malkin, D. Boneh. "Building Intrusion Tolerant Applications". In Proceedings of the 8th USENIX Security Symposium, 1999, pp. 79-91.
- [27] V. J. Hardy. Java 2D API Graphics. Palo Alto, CA: Sun Microsystems Press, 2000.
- [28] W.-H. Ju, and Y. Vardi. "A Hybrid High-order Markov Chain Model for Computer Intrusion Detection". Journal of Computational and Graphical Statistics, Vol. 10, No. 2, pp. 277-295.

Vita

Victor Pimentel de Cerqueira was born in Rio de Janeiro, Brazil on July 31, 1979. He graduated from St. Sebastian High School in Rio de Janeiro, Brazil in 1996 and started college at the Federal University of Rio de Janeiro. He transferred his studies to the University of Tennessee, Knoxville in the Fall of 1999, and received a Bachelor of Science degree in Computer Science in the Spring of 2001. He continued at the University of Tennessee for his graduate studies, and was awarded the Master of Science degree in Computer Science in the Spring of 2003.