To the Graduate Council:

I am submitting herewith a thesis written by Geoffrey Alan Mazeroff entitled "Probabilistic Suffix Models for Windows Application Behavior Profiling: Framework and Initial Results." I have examined the final electronic copy 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 Thomason

Bradley Vander Zanden

Accepted for the Council:

Anne Mayhew

Vice Chancellor and Dean of Graduate Studies

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

Probabilistic Suffix Models for Windows Application Behavior Profiling: Framework and Initial Results

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Geoffrey Alan Mazeroff

December 2004

Acknowledgments

Academically, I would like to thank my research advisors, Drs. Jens Gregor and Michael Thomason for their insight and guidance throughout the work culminating in this thesis. I would also like to thank Dr. Bradley Vander Zanden for participating on my thesis committee. This research was greatly enhanced by the data and domain expertise provided by Drs. James Whittaker and Richard Ford and their team of researchers at Florida Institute of Technology. Finally, I would like to thank Victor de Cerqueira for his work on SPARTA, the graphical user interface for the toolkit developed thus far.

The research presented in this thesis was made possible through funding provided by the Office of Naval Research under grant N00014-01-1-0862.

Personally, I express my deepest thanks to my mother who constantly serves as a source of encouragement, inspiration, and love. I also thank my officemates for their friendship, and for tolerating my frustration when things weren't working. Finally, I would like to thank Gillian Hunt for her selfless dedication, support, and love that she has provided me during these past several months.

Abstract

Developing statistical/structural models of code execution behavior is of considerable practical importance. This thesis describes a framework for employing probabilistic suffix models as a means of constructing behavior profiles from code-traces of Windows XP applications. Emphasis is placed on the inference and use of probabilistic suffix trees and automata with new contributions in the area of auxiliary symbol distributions. An initial real-time classification system is discussed and preliminary results of detecting known benign and viral applications are presented.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Over the past decade the Internet and ultimately the number of networked computers has grown significantly. Aided by the availability of high-speed network connections on college campuses as well as at home and commercial sites through Internet service providers, large amounts of information can be sent and received with low latency. Unfortunately, malicious mobile code (MMC), for example, worms and viruses, thrive in this virtual "playground" of potentially exploitable machines.

Malicious programs, also described as *malware*[14], are designed to move from one computer to the next with intent to modify the compromised system without the consent of the operator[7]. The goals of MMC vary from attempting to compromise as many machines as possible to making targeted attacks (e.g., denial-of-service attacks on specific websites). MMC is an important issue to address in computer security as both

the infection rate and severity of the damage have increased over time[15]. It is important to note that MMC is a broad category including generic malicious applications as well as viruses, worms, and trojans. The specific MMC discussed in this thesis pertains only to the latter categories.

With new variants of viruses emerging so rapidly, antivirus vendors must respond quickly to provide up-to-date protection for their customers. Symantec Corporation, one company that provides virus protection, has definitions/signatures for nearly 68,000 viruses[2]. Given the increased connectivity and the speed and extent of viral propagation, acquiring a system to detect these emerging threats becomes critical. This thesis discusses the framework and initial results of one such system built under the paradigm of being able to detect known as well as unknown threats through application behavior modeling.

1.2 Types of Detection

Systems for protecting computers from MMC are categorized into providing either *misuse* or *anomaly* detection. Misuse detection is based on knowledge of previously observed or known attacks (e.g., virus signature recognition), whereas anomaly detection is based on knowledge of acceptable system behavior (e.g., user/application profiling). Both types of detection have strengths and weaknesses. Misuse detection excels in detecting known threats with a very low probability of classifying benign behavior as malicious (i.e., minimal false positives); however, misuse detection is "reactive" in that

the threats must be made known to it in order for malicious activity to be detected. Anomaly detection excels in detecting new or previously unobserved threats, or more specifically, behavior that does not match known/trained profiles; however, it may be the case that benign applications can exhibit behavior that does not match the detection system's profiles, thereby yielding a potentially higher probability of encountering false positives. In practice, misuse detection is often chosen because of its success in detecting threats and low false positive rates. Nonetheless, anomaly detection is of considerable interest both in practice and in MMC classification research. This thesis focuses on the description of one such method of employing anomaly detection through the modeling of benign application behavior.

1.3 Related Work

The modeling of application behavior, whether benign or malicious, is not a new concept and has been investigated by many others. Although the breadth of related work discussed here is not exhaustive, several publications are of interest. The work of Forrest *et al.*[5] closely resembles the approach described in this thesis. Sequences of UNIX system calls are analyzed to build a database of normal behavior containing sequences of observed system calls. Apap *et al.*[1] use Windows registry data to construct feature vectors of observed behavior. They propose that registry usage is "regular" and that their method of anomaly detection could be used to support an existing intrusion detection system. As a type of machine learning, hidden Markov models (HMMs), is

used by Ourston *et al.*[11] as well as Yeung and Ding[18]. The Ourston *et al.* approach employs HMMs for detecting network intrusions whereas Yeung and Ding's approach uses HMMs for building user profiles based on UNIX system call sequences. Other statistical approaches have been investigated by Schultz *et al.*[13] in the analysis of byte sequences using naive Bayes classifiers. Additionally, DuMouchel and Schonlau propose applying principal component regression to sequences of user commands to detect masquerades[3]. Finally, Giacinto *et al.*[6] use a pattern recognition approach to network intrusion detection based on the fusion of multiple classifiers to provide a better trade off between generalization and false alarm rates.

1.4 Goal and Overview

The goal of this thesis is to discuss a framework for profiling Windows applications using probabilistic suffix models and to present some initial results with regard to detecting MMC. Chapter 2 describes two probabilistic suffix models: the probabilistic suffix tree (PST) and probabilistic suffix automaton (PSA). Chapter 3 discusses the framework for inferring and using the aforementioned models for profiling application behavior. Chapter 4 presents initial classification results as a result of the described methods. Finally, Chapter 5 provides a summary of this thesis and a discussion of future work.

Chapter 2

Probabilistic Suffix Models

2.1 Probabilistic Suffix Trees

2.1.1 Description

A probabilistic suffix tree (PST), as described in [12], is an n-ary tree whose nodes are organized such that the root node gives the unconditional probability of each symbol of the alphabet while nodes at subsequent levels give next-symbol probabilities conditioned on a combination of one or more symbols having been already observed (i.e., a history). The probabilities are relative frequency-count estimates of symbol occurrences in the learning sample(s). PSTs also have a property of *order* corresponding to the depth of the tree, such that a kth-order PST has $k+1$ level(s). To illustrate, Fig. 2.1 shows the third-order PST inferred from the sample "`2 1 1 3 1 3 1 3 3 4`" where symbols "2" and "4" are used uniquely as the start-of-string and end-of-string delimiters. Beginning

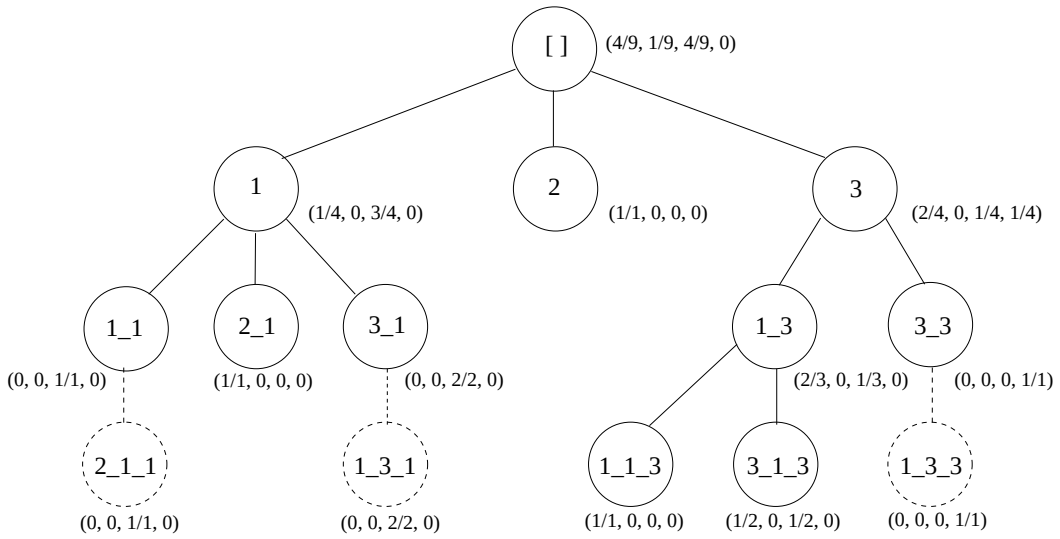

Figure 2.1: Example of full and pruned PST.

with the root node which represents the empty string, each node is a *suffix* of all its children – hence, the model is a *suffix* tree. The root, representing the empty string, is inherently a suffix of all singleton symbols. The numbers in parentheses adjacent to the nodes are the conditional next-symbol probabilities, with the exception that the next-symbol probability distribution for the root node has probability 0 for the unique end symbol "4" because that symbol itself has no next-symbol in the learning sample. One should also note that next-symbol "transitions" jump from one branch to another, not from a parent node to one of its children, because of the suffix format of the node labels.

2.1.2 Inference

The recursive algorithm for inferring a PST is summarized as follows. The inference sample is scanned to determine the probability of each symbol. Then each symbol

with non-zero probability becomes a child node of the root, and for each such node the sample is re-scanned to determine the next-symbol probability distribution. This may create new leaf-nodes and add another level to the PST. The inference recursively adds additional levels by checking each current leaf-node to determine whether new leaf-nodes must be created as its offspring. Information about the symbols preceding the current leaf-node is retained because these symbols may appear in labels of children thereof. A node is added only if the substring of symbols that corresponds to the node label is in the learning sample(s) and if a non-zero next-symbol probability distribution exists for it. Some branches typically die out early while other branches propagate to the maximum depth of the tree. The depth (i.e., order) can be set to a maximum allowable value, or by default, is the length of the input sequence. Because of this property of conditionally adding branches as needed, a PST's *variable memory length* is determined by patterns in the learning data itself.

A recursive bottom-up pruning process may now be carried out to remove leaf-nodes that provide the same statistical information as their parent nodes. Removing these "same-as-parent" nodes eliminates redundancy within the tree structure, thereby creating an attractively parsimonious model. Dashed nodes and lines in Fig. 2.1 indicate pruned nodes and associated branches. The pruning is based on the probability information, not the frequency counts themselves.

2.1.3 PST Operations

The tree structure of the PST model lends itself to several types of analytical or manipulative operations. Such operations will only be mentioned briefly here, as they are beyond the scope of this thesis. The primary role of the PST in this framework is to stochastically represent training data in a tree structure that ultimately is transformed into an analogous Markov chain, a process described later in Section 2.2.1.

Preliminary investigation has been performed in the area of PST *merging*, *reconstruction*, and *comparison*. Multiple PSTs, potentially representing varied or similar learning samples, can be merged into a single tree that represents a PST that could have otherwise been inferred by using all of the individual PSTs' training sequences. This operation allows for direct manipulation of the PSTs themselves without spending additional computational effort to re-infer a larger PST. With regard to reconstruction, as mentioned previously, certain nodes are pruned from the PST if they provide the same stochastic information as their parents. An algorithm has been developed to reconstruct the frequency counts of next-symbols for the deleted nodes, thereby reconstructing or "flooding" nodes in the PST. This technique is useful when performing analytical operations on PSTs, such as comparing two trees. There exist several algorithm variants for finding the distance between two given PSTs. This metric has potential value in finding the degree of similarity or dissimilarity between PSTs. Investigative work in this area is currently being pursued, as this technique may provide some insight into the classification of observed application behavior.

2.2 Probabilistic Suffix Automata

2.2.1 Description

The probabilistic suffix automaton (PSA), as described in [12], is a recurrent discrete-parameter Markov chain of variable-order, and is organized such that transitions exist between states as dictated by non-zero next-symbol probabilities in the PST. The PSA can be thought of as the Markov chain analog of the PST, as it contains the same next-symbol probability information as the tree. The states in the PSA are labeled in the same manner as nodes in the PST: symbols comprising the history (i.e., providing variable memory length) are concatenated by underscores, where the number of symbols comprising the history is at most k, where k is the order of the PST. To illustrate, Figure 2.2 shows an example second-order PSA containing three transient states (i.e., "0", "1", and "2") and five recurrent states. The values on the arcs connecting two given states represent the probability of transitioning from one state to the next. One can also think of the arcs between states being associated with the right-most symbol of the destination state, thus "transitioning on a single symbol." For example, the probability of encountering "1" after having just seen "3" is 0.5, as indicated by the probability on the arc between states "3" and "3_1". Transitions may only occur where arcs are present, otherwise the probability of the given symbol is zero.

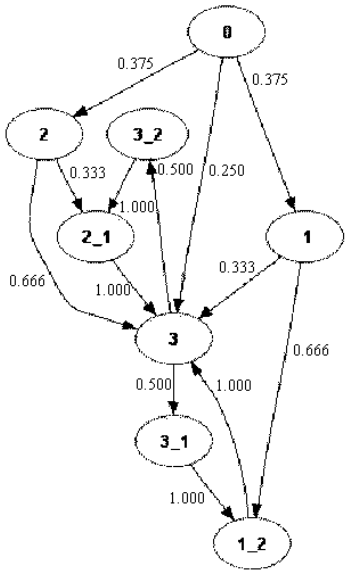

Figure 2.2: Example PSA.

2.2.2 Inference

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. There are *predecessors* rather than parents and *successors* rather than children. 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 PST 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", the intermediate states "1_3" and "3" are created and then connected from the root state to "3", "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, the symbol is appended to the end of the state label and symbols are removed from the *front* of the new label until a state is found that exists in the automaton. For example, state "3_2" in Fig. 2.2 has a non-zero probability of encountering a next symbol of "1"; therefore, a new label is created, "3_2_1" and symbols are removed from the left side until a match is found. Removing the "3" yields "2_1" which is a defined state, so an arc is created between "3_2" and "2_1". 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.

2.2.3 Matching Techniques

Matching samples against a suffix automaton yields the cumulative probability of the given sample being generated by the PSA. The higher the probability of match, the more likely it is that the sample is similar to the sequence(s) used to infer the PST and likewise, the PSA. To match a sample, the current state is set to the "root" state. The

sample is traversed symbol-by-symbol, appending the next symbol onto the previous one. If the state based on the newly created label exists, the probability of transitioning from the previous state to the current state is noted and symbols from the sample continue to be appended. If the node does not exist, however, one symbol at a time is removed from the front of the target state label until a node that does exist in the PSA is encountered. Note that in the worst case the ultimate state achieved after backtracking will be a transient state because this state should contain the next-symbol probability to proceed with the matching algorithm. This worst-case is similar to backtracking to the root node in the PST. For example, to match the sample "1 3 2" against the PSA in Fig. 2.2, the following steps would occur. Step 1: node "1" exists so "3" is appended, 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 the product of these two probabilities: 1/3.

Using the approach stated previously, the probability of match for the entire sequence can be found. If the sequence contains several symbols, N, the cumulative probability of match can be very small, as fractional probabilities are continually multiplied by other fractional probabilities. This issue is addressed by [8] by employing a "sliding window" approach in which the probability of m symbols is calculated, where $m \ll N$. This technique allows a series of probabilities to be computed as the match sample is traversed. The framework presented in this thesis employs this matching technique to ultimately achieve application behavior classification.

2.2.4 PSA Operations

As with the PST, the PSA model lends itself to several types of analytical or manipulative operations. Again, such operations will only be mentioned briefly here, as they are beyond the scope of this thesis. The primary role of the PSA in this framework is to serve as a model of known behavior so that probabilities of given match sequences can be computed to determine how closely the sequences represent the inference data.

Because the PSA is a Markov chain, any mathematical operations that can be used with Markov models apply[9]. Some example Markovian statistics of interest include steady-state distributions and the mean number of transitions between states. The steady-state distributions may be used to determine how frequently a given PSA state is visited in the long run. The mean number of transitions can be used to determine if transitions between two given states are occurring at predictable rates. For example, if a sample causes the PSA to transition from state i to state j in a consistently low number of transitions, and the mean transition value m_{ij} is quite large, this scenario may indicate that the observed sample is getting from state i to state j through some shorter route than the expected one. Additionally, investigative work is being pursued in the area of comparing two PSAs, similar to the operation of comparing two PSTs, as this technique may also provide some insight into the classification of observed application behavior.

Chapter 3

Modeling Application Behavior

3.1 Detection System Context

The application behavior framework presented in this thesis will ultimately be integrated into a Windows-based system developed by collaboration with the Florida Institute of Technology (FIT). Their system called HEAT (Hostile Environment Application Testing) as originally proposed by [16] intercepts low-level system calls made to the operating system, e.g., when a program reads or writes a file, accesses or modifies the registry, dynamically loads runtime libraries, and so forth. HEAT serves as the underpinning for another application layer called Gatekeeper (GK)[4]. See Fig. 3.1. Gatekeeper monitors the information pertaining to the intercepted function calls to determine if the monitored application is exhibiting benign or malicious behavior. Gatekeeper currently uses a genetic algorithm-tuned scoring system to assign threat levels to predefined behaviors. If too many malicious behaviors of a certain nature occur within a certain period, the

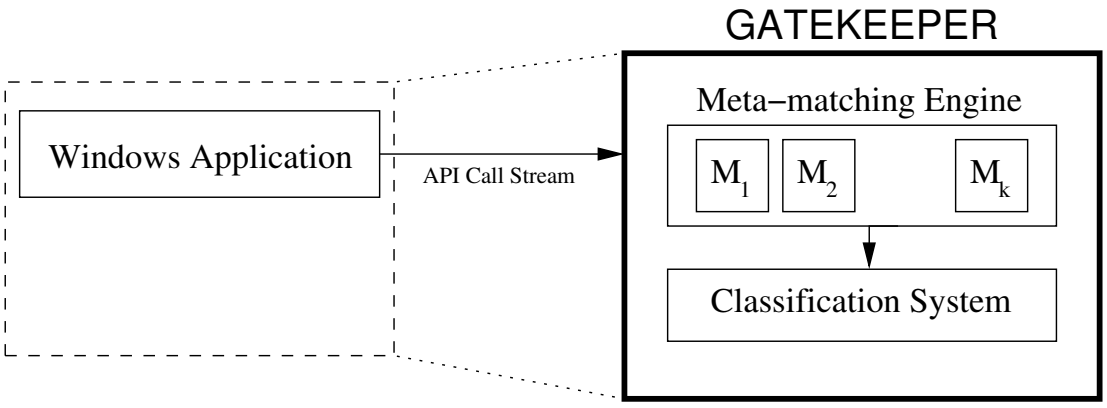

Figure 3.1: Conceptual view of Gatekeeper.

offending application is terminated in an attempt to prevent any further unwarranted actions. Future developments of Gatekeeper include the use of multiple detection engines, including the original score-based approach, the probabilistic suffix model approach described in this thesis, and perhaps other methods to create a "meta-matching engine" that will ultimately decide whether to allow or block application system calls.

The classification framework based on the aforementioned probabilistic suffix models provides an additional classification approach to augment the existing detection capabilities of Gatekeeper. The two approaches differ in that Gatekeeper's current detection technique is based primarily on predefined malicious behaviors. One should note that the system is tuned for detecting malicious *behavior* rather than specific malicious *signatures*, as most virus scanners typically do. The detection technique presented in this thesis is entirely data-driven and assumes no pre-existing knowledge of specific benign or malicious behavior. The goal is to infer a model (or possibly models) of fundamental benign application behavior such that other benign applications exhibit behavior simi-

Table 3.1: Excerpt of log trace information.

PID	`2760`
Timestamp	`19:24:06:510`
Function	`LoadLibrary`
Parameter	`C:\WINNT\ System32\ADVAPI32.DLL`

lar to that of the applications used for model inference, whereas malicious applications exhibit anomalous behavior.

3.2 Model Inference

Gatekeeper currently has the ability to function as a "benign tracer," allowing applications to be monitored for the sole purpose of recording what system call activity takes place during the application's execution. The information made available by Gatekeeper consists of text-based logs consisting of one line for each intercepted system call. Table 3.1 presents some of the information for one such log entry associated with loading a dynamically-linked library.

The current implementation of the PST and PSA inference tools rely on integer-encoded symbols as input. Therefore, the first step in modeling application behavior is to map the appropriate string-based log information into integers. As a first approach used in this initial framework, the intercepted function calls are grouped into three categories – file, registry, and library – as the majority of function calls that applications use during their execution fall into these groups. Table 3.2 gives the specific mapping used for model inference.

Table 3.2: Integer mapping definition.

1, 2	Sequence-start and Sequence-end
3	File Operations
4	Registry Operations
5	Library Operations

Once the log files designated for training have been converted into integer sequences, they are concatenated together to form one long training sequence. Note that concatenation does not affect the PST inference accuracy because special separators used in the concatenation process indicate that the symbol frequency counts should not overlap from the end of one sequence to the beginning of the next. After the PST has been inferred, it is in turn used to infer a corresponding PSA. Finally, the PSA is used as a model of learned behavior so that other application traces (as integer sequences) can be matched against it to compute probabilities of match.

3.3 Auxiliary Symbol Distributions

3.3.1 Description

Although the initial symbol (i.e., alphabet) choice seems to cover the broad set of operation types that applications employ, it is important to note the domain being modeled. Given the implementation and protocols of system call usage defined by the Windows operating system, certain patterns of system call sequences become evident in both benign and malicious applications. For example, for a file to be modified, it must be opened, written to, and finally closed. This open-use-close pattern also occurs with

respect to registry use. Because this behavior is common to both benign and malicious applications, using the *auxiliary* information provided for each function call (e.g., its parameters) can allow more specific information to be integrated into the model. For example, this auxiliary information indicates specific libraries that are loaded, registry keys that are used, directories that are managed, and so forth. Such information would be stored for every node in the PSA so that every symbol context has a corresponding auxiliary context as well. In other words, given a symbolic context (i.e., state in the PSA), the probability of encountering a given piece of auxiliary data (e.g., registry key, library filename, etc.) can be determined. The probability distributions can vary for each node because different information (e.g., registry keys used, files opened, etc.) may be different for each context as it is observed in the training data.

The additional information is deemed as *auxiliary* because it refines the alphabet without changing the underlying probability model. One can think of this auxiliary information as an added layer of conditional probability – for example, the probability of loading a specific library given that a certain context of file, registry, or library operations has occurred. An equivalent symbol mapping can be achieved by designating symbols to account for the context. For example, if symbol "3" corresponds to file operations, symbol "30" could indicate file operations in the user's area, "31" could indicate file operations in the system directory, "32" could indicate file operations in a temporary directory, and so on. The drawback to creating additional alphabet symbols is that the degree of the PST increases, thereby increasing the number of branches/children a

given node can have. This in turn increases the computational cost of inferring as well as using larger, more complex models. By using a small but coarse primary alphabet (i.e., broad categories), the auxiliary information can accompany the smaller models without unnecessarily inflating the model size.

With regard to file and registry key auxiliary data, an additional layer of conditional probability is imposed. Either based on the function name or its parameters, a given file or registry function can execute in one of the following contexts: access, modification, deletion, or creation. In other words, the auxiliary information ultimately provides the probability of a specific piece of data (e.g., registry key) working within a specific execution context (e.g., deletion) given that a certain sequences of symbols has occurred. This added layer of conditional probability exploits a feature of the problem domain in that benign applications work with specific auxiliary data in a certain context, whereas malicious applications are more likely to work in another context. For example, a benign application such as a text editor would not be expected to delete all of the files in the system directory whereas a malicious application may.

3.3.2 Inference

The algorithm for inferring the auxiliary symbol distributions (ASDs) is summarized as follows. For each state in the PSA, the integer-based training sequences are traversed to find all occurrences of the symbols corresponding to the state's label. These locations are used to find the corresponding function call data in the original logs so that the parameter information can be extracted. A simple count is kept for the number of

occurrences of the given parameter. If the parameter is encountered again in another training sequence (for the same state), the count is incremented, thereby creating a distribution of frequency counts of occurrences for auxiliary data items observed for the context designated by the state label. File and registry operations are partitioned into the four previously mentioned execution contexts, thereby creating four unique distributions for the given state (e.g., items accessed, modified, deleted, and created). The final ASD model consists of a distribution (or distributions in the case of file and registry operations) for each node in the PSA.

3.3.3 Matching

The ASD model corresponds directly to the PSA as there are distributions for each state. As the probability of match is computed as defined in Section 2.2.3, the probability of encountering a given auxiliary data item is computed for the current state. The probability of auxiliary data is simply the ratio of the number of times the given item has been observed to occur to the number of total occurrences of all auxiliary items for the current state. For example, if a given library has been observed to occur in 100 out of 150 times, the probability assigned is 0.667; however, if a given library has never been observed in the ASD inference process for the current state, a probability of zero is assigned.

3.4 Classification Approach

Once the models of benign applications (i.e., PSA and ASD) are in place, test sequences representing both benign and malicious applications can be matched against them to determine the probability of match. As mentioned in Section 2.2.3, a sliding window technique is used rather than calculating the cumulative probability of match. In this initial framework, a sliding window size of one symbol is used, thus giving $n-1$ probability values for a sequence of length n. Given the close coupling of the PSA and ASD models, the product of the match probabilities computed from each of the models is the probability value used in classification, rather than treating the two probabilities separately. Initial experiments showed that the match probabilities exhibit high variability throughout the sequence, indicating that some type of smoothing computation would be necessary to capture the overall probability trend. For this particular framework, an exponentially-weighted moving average (EWMA) was used so that the amount of emphasis placed on recent or past events could be varied. Formally, the EWMA is defined as

$$y_i = \lambda x_i + (1-\lambda) y_{i-1}$$

where x_i is the ith probability of match, y_i is the ith exponentially weighted probability of match, and λ is the EWMA parameter such that $0 \leq \lambda \leq 1$[10]. If more emphasis is to be placed on immediate or more recent events, larger λ values are used.

Sequence classification is performed based on an empirically-derived threshold. The

initial experiments and techniques presented in this thesis are based on the data being partitioned into four groups: benign training, benign testing, viral training, and viral testing. Each of the sequences in the viral training category are used to determine the threshold. First, the exponentially-weighted probabilities for each viral training sequence are computed. Second, the mean EWMA value for each sequence is calculated. Lastly, the maximum EWMA mean is used as the threshold τ. This technique guarantees that the models will, at worst, identify all viral training samples as malicious. Classification of an arbitrary sequence is performed by observing if $y_i > \tau$, where y_i is the exponentially-weighted probability. If the threshold is exceeded, the sequence is labeled as *malicious*; otherwise, the sequence is labeled as *benign*. Formally, if a benign sequence is misclassified, a *false positive* occurred (e.g., a benign application labeled as malicious); likewise, if a malicious sequence is misclassified, a *false negative* occurred (e.g., a malicious application labeled as benign). The performance of the techniques used in the initial framework presented here are based on the false positive rate and the true negative rate, or in other words the number of benign applications erroneously stopped and the number of malicious applications correctly stopped.

Chapter 4

Experiments and Results

4.1 Datasets

4.1.1 Benign and Malicious Samples

In this initial experiment, two classes of applications are monitored – benign and malicious (i.e., viral). The *benign* application set consists of ten runs each of five commonly-used Microsoft applications: Word, Excel, PowerPoint, Outlook, and Internet Explorer. Each of these runs was created by performing typical yet unique tasks that would be carried out in various computing environments, such as personal, academic, and commercial. Applications from the same vendor (i.e., Microsoft) were chosen so that application variability would be minimized via the use of similar implementation schemes, commonly-used libraries, and so forth. See Appendices A and B for descriptions of the benign training and testing sequences respectively. The *malicious* application set con-

sists of 223 Windows viruses that were known to be "in the wild" for specific months, ranging from December 2002 to March 2004[17]. This set of viruses contains several well-known threats as well as some variants for specific viruses. See Appendices C and D for listings of the viruses in the training and testing sets respectively.

4.1.2 Training and Testing Samples

The benign and malicious application traces are both partitioned into training and testing groups. The *benign* set is divided such that there are five runs for each of the five applications in each set, yielding 25 training traces and 25 testing traces. The *malicious* set is grouped in the same manner as the co-researchers at FIT, where 73 viruses from the December 2002 wild-list are designated as training and the remaining 150 viruses from the January 2003 to March 2004 wild-lists are used for testing[4]. The motivation behind this partitioning is to provide training samples from an older group of viruses to determine how well the detection system is able to detect progressively newer threats. The same approach is adopted here so that future comparisons of detection effectiveness can be made.

4.1.3 Model Descriptions

Each of the 25 training samples are re-mapped into integer symbols and are concatenated to form a single inference sample as discussed previously in Chapter 3. This sequence is used to infer a PST and PSA, thus creating a model of benign behavior for the five monitored applications. Specifically, the inference sequence contains 203,820 symbols,

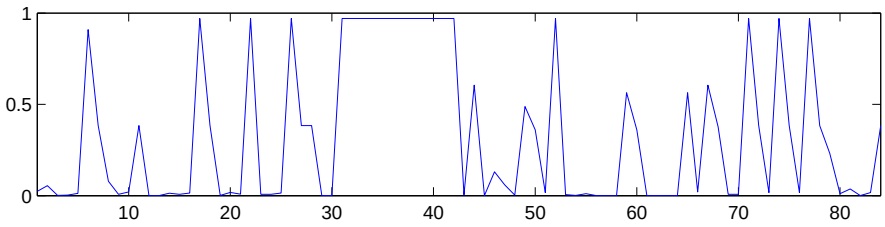

Figure 4.1: Excerpt of the probability of match for a run of Internet Explorer.

which is then used to infer a second-order PST containing 15 nodes. The PST is then used to infer a second-order PSA containing 16 states. Although not discussed here, previous experiments using application traces from a earlier version of Gatekeeper have shown that model order does affect the classification abilities; however, for the conditions of the framework presented here, preliminary trials showed that desirable classification results could be achieved without a large history (i.e., high model order), but using some history proves to be effective (e.g., still employing a variable-memory model).

4.2 Model Usage

Once the PSA is in place, each of the individual traces from the benign and malicious testing sets are matched against the PSA. For example, an excerpt of the probability of match values for a benign testing sequence for Microsoft Internet Explorer is shown in Fig. 4.1 and a similar excerpt of the W32@Yaha.U virus is shown in Fig. 4.2.

Next, each of these match probability sequences are smoothed using an exponentially-weighted moving average with a parameter of $\lambda = 0.001$, thereby giving more emphasis on past events and less on recent events. The mean EWMA probability is calculated

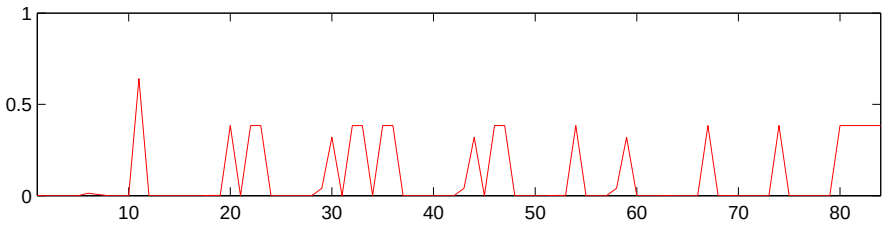

Figure 4.2: Excerpt of the probability of match for the W32@Yaha.U virus.

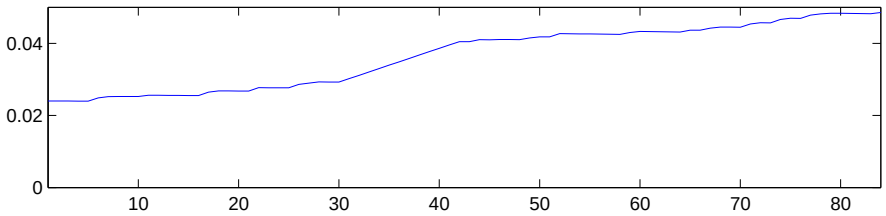

Figure 4.3: Excerpt of the EWMA probability for a run of Internet Explorer.

for each of the 73 malicious training samples to ultimately find the highest EWMA mean that will be used for classification. For the data in this experiment the threshold is calculated to be $\tau = 0.02$. Figs. 4.3 and 4.4 show the exponentially-weighted probabilities.

For these particular samples, Internet Explorer does not fall below the threshold and is therefore classified as *benign*; however, W32@Yaha.U fails to exceed the threshold and is therefore classified as *malicious*.

4.3 Classification Results

Each of the sequences designated for testing are matched, smoothed, and classified in the same manner. Each of the 25 benign testing sequences have EWMA probabilities that continually remain above τ and are correctly classified as *benign*. Likewise, each of

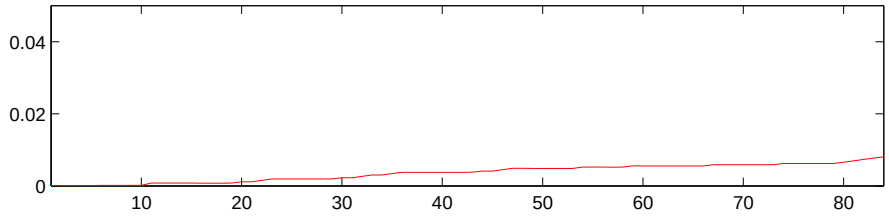

Figure 4.4: Excerpt of the EWMA probability for the W32@Yaha.U virus.

the 150 viral testing sequences have EWMA probabilities that are always below τ and are correctly classified as *malicious*. Overall, this initial detection framework provides optimal classification: a false positive rate of 0% and a true negative rate of 100%.

It is important to note that these ideal classification results, although very promising, may only apply to the specific data and modeling techniques used in this experiment. In practice, a general solution is needed that is able to account for more varied types of training and testing data without significantly raising the false positive rate or lowering the true negative rate. If such results were to occur, the end-user would likely not use the detection system because of benign applications being erroneously terminated, or would not be adequately protected from potential threats, respectively. It therefore becomes necessary to find a balanced system that can distinguish unobserved benign behavior from true malicious behavior without disrupting the end-user's computing experience.

Chapter 5

Summary and Future Work

This thesis presented an initial framework for profiling Windows applications using probabilistic suffix models. These models are trained using benign application traces so that applications exhibiting unobserved behavior can be deemed anomalous. Initial classification results show that using multiple runs of a small set of applications (to provide statistical reinforcement) can provide accurate classification of both benign and malicious sequences.

As mentioned previously, misclassifications may occur as the set of training and testing applications becomes broader (i.e., more than five Windows applications, or additional samples even within those five). Further refinement of data encoding with regard to function-to-integer mapping as well as auxiliary distribution handling can be used to possibly lead to a more general stochastic model of typical benign behavior while still providing acceptable classification results. As this occurs, the amount of training

and testing data will need to increase to determine the capabilities of the system. The initial results presented in this thesis are based on a relatively small training and testing sets, and more sequences will be required to ensure consistent performance. In addition to the probabilistic suffix model operations currently being investigated (as mentioned in Chapter 2), several statistical approaches pertaining to using match probabilities are being studied. Other types of data modeling tools and classification techniques are also being pursued. Lastly, a more generalized version of the detection system presented here will be integrated into the Gatekeeper software package assist in providing real-time threat detection.

Bibliography

Bibliography

[1] F. Apap, A. Honig, S. Hershkop, E. Eskin, and S. Stolfo. Detecting malicious software by monitoring anomalous Windows registry accesses. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection*, pages 16–18, Zurich, Switzerland, 2002. Interface Foundation of North America.

[2] Symantec Corporation. Symantec Security Response. http://securityresponse.symantec.com/avcenter, accessed August, 2004.

[3] W. DuMouchel and M. Schonlau. A comparison of test statistics for computer intrusion detection based on principal components regression of transition probabilities. In *Proceedings of the 30th Symposium on the Interface: Computing Science and Statistics*, pages 404–413, Minneapolis, 1999. Springer-Verlag Zurich.

[4] R. Ford, M. Wagner, and J. Michalske. Gatekeeper II: New approaches to generic virus prevention. In *Proceedings of the 14th Virus Bulletin International Conference*, 2004. To appear.

[5] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff. A sense of self for Unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 120–128, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[6] G. Giacinto, F. Roli, and L. Didaci. Fusion of multiple classifiers for intrusion detection in computer networks. *Pattern Recognition Letters*, 24:1795–1803, 2003.

[7] R. Grimes. *Malicious Mobile Code: Virus Protection for Windows*. O'Reilly and Associates, Sebastopal, CA, 2001.

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

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

[10] D. Montgomery. *Introduction to Statistical Quality Control*. John Wiley and Sons, New York, 1991.

[11] D. Ourston, S. Matzner, W. Stump, and B. Hopkins. Applications of hidden Markov models to detect multi-stage network attacks. In *Proceedings of the 36th Hawaii International Conference on System Sciences*, pages 334–343, Big Island, HI, 2002. IEEE Computer Society Press.

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

[13] M.G. Schultz, E. Eskin, E. Zadok, and S.J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 38–49, Oakland, CA, 2001. IEEE Computer Society Press.

[14] E. Skoudis and L. Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall, Upper Saddle River, NJ, 2004.

[15] M. Wagner. Behavior oriented detection of malicious code at run-time. Master's thesis, Florida Institute of Technology, 2004.

[16] J.A. Whittaker and A. de Vivanco. Neutralizing Windows-based malicious mobile code. In *Proceedings of the ACM Symposium on Applied Computing*, pages 242–246, Madrid, 2002. ACM Press.

[17] WildList Organization International. Wildlist. http://www.wildlist.org, accessed August, 2004.

[18] D-Y Yeung and Y. Ding. Host-based intrusion detection using dynamic and static behavioral models. *Pattern Recognition*, 30:229–243, 2003.

Appendices

Appendix A

Benign Training Sample Descriptions

`Excel_DataentrySave` – Entered data into rows and columns, then saved to My Documents.

`Excel_LaunchExit` – Launched Excel, then exited.

`Excel_LoadChartSave` – Loaded existing worksheet and made a chart from the data. Saved the worksheet with the chart (under the same filename).

`Excel_LoadPrint` – Loaded existing worksheet and printed it to the default printer.

`Excel_LoadWorksheetSave` – Loaded saved worksheet, modified some cell data, then saved the worksheet (under the same filename).

`IE_Amazonnav` – Logged into amazon.com and looked for previous invoices (uses cookies).

`IE_GooglesearchNav` – Used google.com for two searches and followed top-most link for each search.

`IE_LaunchExit` – Launched Internet Explorer, then exited.

`IE_NavJavascript` – Loaded a page with a Javascript countdown timer.

`IE_SymantecnavPrint` – Searched symantec.com for information on a recent virus, then printed the page to the default printer.

`Outlook_Calendarmanip` – Viewed the calendar and added an entry/appointment.

`Outlook_Emaildraft` – Started a reply to an email in the Inbox and saved it as a draft.

`Outlook_LaunchExit` – Launched Outlook, then exited.

`Outlook_Printcalendar` – Viewed the calendar and printed the weekly-view to the default printer.

`Outlook_Taskmanip` – Viewed the task list, added two items. "Checked off" one item, then deleted it.

`PPT_LaunchExit` – Launched PowerPoint, then exited.

`PPT_LoadPrint` – Loaded existing presentation and printed "handouts" to the default printer.

`PPT_LoadShow` – Loaded existing presentation, then performed the "slide show."

`PPT_LoadTransSave` – Loaded existing presentation, added transitions between each slide, and saved presentation (under the same filename).

`PPT_MakepresntSave` – Made a five-slide presentation, then saved to My Documents.

`Word_FormattextSave` – Entered some text and formatted it in various ways. Saved document to My Documents.

`Word_LaunchExit` – Launched Word, then exited.

`Word_LoadPrint` – Loaded existing document and printed it to the default printer.

`Word_LoadWordartSave` – Loaded existing document, added WordArt, then saved document (under the same filename).

`Word_TableSave` – Added a table to an empty document, entered data, then saved document to My Documents.

Appendix B

Benign Testing Sample Descriptions

`Excel_Chartformat` – Loaded existing worksheet. Plotted data. Formatted the axes, plot background fill pattern, and line color. Saved to My Documents.

`Excel_DataentryCellnames` – Entered three columns of data and "named the columns (cell range names). Saved to My Documents.

`Excel_Datavalidation` – Loaded existing worksheet, added a data validation rule for a fourth column. (Makes a pull-down list of valid entries, and shows customizable error if user enters invalid data.) Saved to My Documents.

`Excel_Formulas` – Loaded existing worksheet. Inserted some formulas. Activated the Formula Auditing toolbar, then used a tool to show dependencies between cells with formulas. Added a watch to a cell to see how its contents are computed. Saved to My Documents.

`Excel_Subtotals` – Loaded existing worksheet, then used the "subtotals feature to calculate running totals for columns/rows. Saved to My Documents.

`IE_DownloadPics` – Navigated to digitalblasphemy.com and downloaded two background images to My Documents.

`IE_DownloadZip` – Downloaded a Winzip file from my website (IE displayed dialog box prompting for action).

`IE_Flash` – Navigated to a Flash-enabled site (uses browser plugin to handle Flash files).

`IE_Homepage` – Navigated to UTKCS department website and set it as the homepage.

`IE_Newssites` – Navigated to cnn.com, foxnews.com, and reuters.com and clicked on the top headline for each website.

`Outlook_Archive` – Archived Outlook data to My Documents subdirectory.

`Outlook_Contacts` – Added contact to the address book, then searched for that contact.

`Outlook_Customize` – Customized appearance of Outlook startup (colors, fonts, etc.).

`Outlook_Notes` – Edited existing "note" and added a new one.

`Outlook_Recurtasks` – Set up three recurring tasks with various properties (importance, percent done, status, etc.).

`PPT_ChartDiagram` – Loaded existing presentation. Inserted chart and diagram on two new slides. Saved to My Documents.

`PPT_PicsRearrange` – Loaded existing presentation. Inserted two pictures (JPGs), then rearranged some slides. Saved to My Documents.

`PPT_Saveasweb` – Loaded existing presentation. Saved as web page.

`PPT_SlideMaster` – Loaded existing presentation. Altered slide master to have a different font. Saved to My Documents.

`PPT_SlidenotesPrint` – Loaded existing presentation. Added presentation notes for each slide, then printed slides (presentation notes format). Saved to My Documents.

`Word_Inserts` – Loaded existing document. Inserted a "diagram and edited its text. Inserted hyperlink. Saved to My Documents.

`Word_Pagesetup` – Loaded existing document. Changed page orientation (landscape), margins, and added page border. Reformatted the diagram to fit within the page, then did a print-preview. Saved to My Documents.

`Word_Textformat` – Loaded existing document. Changed font color, size, name, and spacing. Enabled automatic page numbering. Saved to My Documents.

`Word_Mailinglabels` – Created mailing labels, then printed them. No save performed.

`Word_Textentry` – Typed some text. Misspelled some words intentionally (to use automatic spell-check feature). Made a numbered list with AutoFormat. Spell-checked document. Saved to My Documents.

Appendix C

Viral Training Sample Listing

Viruses listed by the names assigned by Symantec Corporation[2].

```
Happy99.Worm
PrettyPark.Worm
W32.Aliz.Worm
W32.Anset.B.Worm
W32.Anset.C.Worm
W32.Anset.Worm
W32.Aplore@mm
W32.Apost.Worm@mm
W32.Badtrans.B@mm
W32.Badtrans.get@mm
W32.Benjamin.Worm
W32.Blebla.B.Worm
W32.Brid.A@mm
W32.Bugbear@mm
W32.Cervivec.A@mm
W32.Chir@mm
W32.Choke.Worm
W32.Datom.Worm
W32.ElKern.3326
W32.ElKern.3587
W32.ElKern.4926
W32.ExploreZip.L.Worm
W32.FBound.gen@mm
W32.Frethem.A@mm
W32.Frethem.L@mm
W32.Gibe.A@mm
W32.Gokar.A@mm
W32.Goner.A@mm
W32.HLLW.Acebo
W32.HLLW.Bymer
W32.HLLW.GOP@mm
W32.HLLW.Hai
W32.HLLW.Oror.B@mm
W32.HLLW.Qaz.A
W32.HLLW.Winevar
```

W32.Higuy@mm
W32.Hybris.A
W32.Hybris.B
W32.Hybris.C
W32.Hybris.D
W32.Klez.A@mm
W32.Klez.B@mm
W32.Klez.C@mm
W32.Klez.D@mm
W32.Klez.E@mm
W32.Klez.G@mm
W32.Klez.H@mm
W32.Kriz
W32.Maldal.C@mm
W32.Maldal.E@mm
W32.Maldal.F@mm
W32.Mylife.A@mm
W32.Mylife.B@mm
W32.Mylife.F@mm
W32.Mylife.G@mm
W32.Mylife.J@mm
W32.Myparty@mm
W32.Naked@mm
W32.Navidad.16896
W32.Navidad
W32.Nimda.A@mm
W32.Pinfi
W32.Prolin.Worm
W32.Sircam.Worm@mm
W32.Supova.Worm
W32.Yaha.C@mm
W32.Yaha.D@mm
W32.Yaha.E@mm
W32.Yaha.G@mm
W32.Yaha@mm
W32.Zoek.E@mm
W95.MTX

Appendix D

Viral Testing Sample Listing

Viruses listed by the names assigned by Symantec Corporation[2].

January 2003 WildList

```
W32.ExploreZip.L.Worm
W32.Frethem.K@mm
W32.Galil@mm
W32.HLLP.Handy
W32.HLLW.Lioten
W32.Opaserv.G.Worm
W32.Opaserv.K.Worm
W32.Opaserv.Worm
W32.Supova.E.Worm W32.Yaha.J@mm
W32.Yaha.K@mm
```

February 2003 WildList

```
W32.Supova.D.Worm
W32.Yaha.L@mm
```

March 2003 WildList

```
W32.Gibe.B@mm
W32.HLLW.Lovegate.C@mm
W32.HLLW.Oror@mm
```

April 2003 WildList

```
W32.Bibrog.C@mm
W32.Ganda.A@mm
W32.Hawawi.D.Worm
W32.HLLW.Lovgate.F@mm
W32.HLLW.Lovgate.G@mm
W32.HLLW.Nebiwo.Q
W32.HLLW.Nebiwo.R
W32.Nicehello@mm
W32.Yaha.P@mm
W32.Yaha.Q@mm
```

May 2003 WildList

W32.HLLW.Deloder
W32.HLLW.Fizzer@mm
W32.HLLW.Lovegate.B@mm
W32.Navidad.16896
W32.Opaserv.G.Worm

June 2003 WildList

W32.Alhem.A@mm
W32.Beast.41472
W32.Bugbear.A@mm
W32.Bugbear.B@mm
W32.Femot.Worm
W32.HLLW.Fizzer@mm
W32.HLLW.Lovgate.I@mm
W32.HLLW.Lovgate.J@mm
W32.HLLW.Lovgate.K@mm
W32.HLLW.Lovgate.L@mm
W32.HLLW.Magold@mm
W32.HLLW.Redist@mm
W32.Hawawi.D.Worm
W32.Kitro.C.Worm
W32.Nolor@mm
W32.Opaserv.J.Worm
W32.Sobig.C@mm
W32.Sobig.D@mm
W32.Yaha.U@mm

July 2003 WildList

W32.HLLW.Magold@mm
W32.HLLW.Purol
W32.HLLW.Winur
W32.Jeefo
W32.Mapson.Worm
W32.Mylife.M@mm
W32.Sobig.E@mm
W32.Yaha.T@mm

September 2003 WildList

W32.Blaster.B.Worm
W32.Blaster.C.Worm
W32.Blaster.E.Worm
W32.Blaster.Worm
W32.HLLW.Warpigs.B
W32.Israz@mm
W32.Mimail.A@mm
W32.Opaserv.P.Worm
W32.Randex.D
W32.Sobig.F@mm
W32.Swen.A@mm
W32.Viveal@mm
W32.Welchia.Worm

October 2003 WildList

W32.Dumaru.C@mm
W32.Dumaru.H@mm
W32.HLLW.Torvel@mm
W32.Inmota.Worm
W32.Mimail.C@mm
W32.Mimail.G@mm
W32.Randex.E
W32.Sober@mm

November 2003 WildList

W32.Mimail.E@mm
W32.Mimail.F@mm
W32.Mimail.G@mm
W32.Mimail.I@mm
W32.Mimail.J@mm

December 2003 WildList

W32.Mimail.M@mm
W32.Sober@mm
W32.Sober.C@mm

January 2004 WildList

W32.Cissi.A@mm
W32.Darker.Worm
W32.Dumaru.Y@mm
W32.HLLW.Darby
W32.Jitux.Worm
W32.Mimail.L@mm
W32.Mimail.P@mm
W32.Mimail.Q@mm
W32.Mimail.S@mm
W32.Mydoom.A@mm
W32.Mydoom.B@mm
W32.Scold@mm
W32.Sober.B@mm

February 2004 WildList

W32.HLLW.Lovgate@mm
W32.HLLW.Lovgate.E@mm

March 2004 WildList

W32.Beagle.C@mm
W32.Beagle.D@mm
W32.Beagle.E@mm
W32.Beagle.F@mm
W32.Beagle.G@mm
W32.Beagle.H@mm
W32.Beagle.I@mm
W32.Beagle.J@mm
W32.Beagle.K@mm
W32.Beagle.N@mm
W32.Beagle.P@mm
W32.Beagle.Q@mm
W32.Beagle.R@mm
W32.Beagle.S@mm
W32.Beagle.T@mm
W32.Beagle.U@mm
W32.Blackmal@mm
W32.HLLW.Capsid
W32.HLLW.Doomjuice.B
W32.HLLW.Lovgate.Q@mm
W32.HLLW.Lovgate.U@mm

W32.HLLW.Raleka
W32.Mydoom.F@mm
W32.Mydoom.G@mm
W32.Mydoom.H@mm
W32.Netsky.B@mm
W32.Netsky.C@mm
W32.Netsky.D@mm
W32.Netsky.E@mm
W32.Netsky.F@mm
W32.Netsky.H@mm
W32.Netsky.J@mm
W32.Netsky.K@mm
W32.Netsky.L@mm
W32.Netsky.M@mm
W32.Netsky.N@mm
W32.Netsky.O@mm
W32.Netsky.P@mm
W32.Netsky.Q@mm
W32.Netsky.T@mm
W32.Sober.D@mm
W32.Welchia.B.Worm
W32.Welchia.C.Worm

Vita

Geoffrey Alan Mazeroff was born to Paul and Sharon Mazeroff in Warrenburg, Missouri on December 28, 1979. He graduated from Riverside High School in Greenville, SC in 1997 and completed his undergradute work at Furman University in 2001. After receiving two bachelor's degrees, a Bachelor of Arts in Music and a Bachelor of Science in Computer Science, he began his graduate work at The University of Tennessee, Knoxville. He was awarded the Master of Science degree in Computer Science in Fall of 2004 and is currently pursuing a doctorate in this field.