Generalized Anomaly Detection Model for Windows-based Malicious Program Behavior

Xin Tang, Constantine N. Manikopoulos, and Sotirios G. Ziavras
(Corresponding author: Sotirios G. Ziavras)

Electrical and Computer Engineering Dept, New Jersey Institute of Technology
Newark, NJ 07102, USA. (Email: ziavras@adm.njit.edu)

(Received June 24, 2007; revised and accepted Apr. 9, 2008)

Abstract

In this paper we demonstrate that it is possible in general to detect Windows-based malicious program behavior. Since S. Forrest et al. used the N-grams method to classify system call trace data, dynamic learning has become a promising research area. However, most research works have been done in the UNIX environment and have limited scope. In Forrest's original model, "Self" is defined based on a normal process whereas "Non-Self" corresponds to one or two malicious processes. We extend this technique into the Windows environment. In our model, "Self" is defined to represent the general pattern of hundreds of Windows program behaviors; "Non-Self" is defined to represent all program behaviors that fall out of norm. Because of the difficulty in collecting program behavior, insufficient research results are available. We collected around 1000 system call traces of various normal and malicious programs in the Windows OS. A normal profile was built using a Hidden Markov Model (HMM). The evaluation was based on the entire trace. Our classification results are promising.

Keywords: Intrusion detection, Markov process, malicious codes, program behavior

1 Introduction

1.1 Motivations and Objectives

Intrusion detection techniques focus on two schemes: the misuse and anomaly schemes. In misuse detection, the Intrusion Detection System (IDS) gathers information and compares with known attack (malicious code) signatures. For example, the signature for the Chernobyl/CIH virus used by one Anti-virus vendor is the hexadecimal segment [7]:

```
E800 0000 005B 8D4B 4251 5050
0F01 4C24 FE5B 83C3 1CFA 8B2B
```

In anomaly detection, the IDS defines a profile for normal (legal) activities; any deviation from this profile is considered as a potential attack (malicious activity). By information collecting, there are two detection approaches for malicious code. One is static analysis that collects program information regarding the code or the binary without actually executing the code. Another is dynamic analysis that collects program information by actually executing the code. The information that is collected by executing the code is collectively known as program behavior. The objective of this paper is to show that it is possible to detect malicious program behavior and to present a generalized anomaly detection model for Windows-based malicious program behavior. With the universal use of computer infrastructure in the government, army, financial institutions, educational institutions and other organizations, ever increasing important information is stored and exchanged between computer systems. Therefore computer security has become a critical issue. A broad category of security threats represents malicious code (or malicious executables; either term is used in the paper based on context). Therefore containing the damage caused by malicious code is of great value to computer infrastructure and our society. To achieve this goal, two aspects must be considered. One is currently known malicious codes that are usually targeted by misuse detection systems, like current Anti-Virus systems. The second aspect is the capability to predict and prevent new malicious code.

1.2 Problem Statement

Since S. Forrest et al. [2, 4] pioneered relevant research with UNIX sequential system call traces, research on run-time program behavior has become a promising area. However, due to difficulties in data collection, results are limited. They proposed the N-grams method, where a profile of normal behavior is built by enumerating all unique, contiguous, length-N sequences that occur in training data. In the classification process, unmatched sequences are labeled as anomalous. They employed the "Self" concept from biology. Matched N-grams are classified as "Self"; unmatched N-grams are "Non-Self". They formed normal data by repeatedly collecting identical normal processes/programs; abnormal data by repeatedly collecting abnormal processes/programs. Essentially, one detection model was built for each normal and corresponding malicious program. Because in normal data there are basically similar (identical or almost identical) traces, it is easy to learn "Self" from similar traces. On the other hand, if we get a new instance to classify, we need to know what it is and which reference model to compare it to. In biology, objects and subjects could be healthy liver cell and liver cancer, or healthy lung cell and lung cancer, etc. If a new instance comes in, we usually know where it is from, and which model we should refer to. In the computer world, things may be more complicated. We do not know what a new instance is and which model we should refer to. So a problem is raised. Can we identify whether a program activity is benign or malicious? We think it is possible. Not only the specific but also the overall characteristics of individual benign program activities and malicious program activities are different. Also, in most cases we just need to know if it is benign or malicious, not necessarily its exact kind of program activity (cell).

Malicious attacks vary. Most attacks are initialed by remote systems through the network. However, for most of the malicious attacks, especially those that cause severe damage to computer systems, damage eventually is inflicted to a target system by executing binaries. So monitoring execution profiles could be an effective approach to predict and prevent malicious attacks. The program behavior data which is collected from execution profiles could be a better source in distinguishing between benign and malicious codes. Besides that, we make here some reasonable conclusions about the general characteristics of benign and malicious codes. Normal programs used by ordinary users are usually well written (by software vendors), whereas malicious programs are usually written by individuals and usually have more errors. But things may not always keep the same. It is reported that more and more European malicious code writers hold a graduate degree. These could be well trained and sophisticated writers.

An attacker trying to access a computer system is similar with a thief trying to break into a house, whilst a legal user trying to access a computer system is similar with the owner entering his own house. Thieves usually are not familiar with the environment. Malicious programs will query more system and user information. Malicious programs may focus more on registry access, file system access, backdoor attempts and other anomalous attempts than normal ones.

We extended S. Forrest's work to generalized models based on the Windows environment. Instead of repeatedly collecting identical process/program data, we collected data from different programs. We built a normal profile based on a group of these programs. We collected data from 493 normal Windows programs and 525 abnormal Windows programs. To the best of our knowledge, our data is by far the largest collection of program behavior traces of both benign and malicious codes.

1.3 System Architecture

A large set of executables were obtained from commonly available sources to provide the data to our approach. Program behavior data was collected by executing the programs under the Windows OS. This is a novel research that no one has done before. Currently, there is no suitable dataset available. No real world malicious data was collected, because malicious executables may infect or crash the computer system. So a special testbed, Microsoft Virtual PC, was implemented to limit the damage of malicious executables. Malicious program behavior data was collected in the Microsoft Virtual PC, whereas normal program behavior data was collected in the regular Windows OS. After statistical pre-processing, features were selected and extracted. The dataset was further split into training and test sets. The training set was used to generate the normal profile based on the Hidden Markov Model. The test set was used to do validation. The final step was to feed test data into a classifier and get detection results.

1.4 Roadmap of this Paper

This paper is organized in the following manner. Section 2 discusses related research works on program behavior detection. Section 3 discusses malicious executables and also presents the data collection process. Section 4 presents a forward detection model: Hidden Markov Model. We conclude the paper in Section 5.

2 Related Work

D. E. Denning [1] established the theoretical foundation for anomaly detection by proposing a general-purpose model for anomaly IDS in 1986. The model is based on the assumption that malicious activities are different from normal activities and could be detected by monitoring system audit information. The model built a framework to implement anomaly detection. It is independent of any particular system, application environment, system vulnerability, or type of intrusion. In 1994, J. O. Kephart [6] first designed the framework for a computer immune system which is alike vertebrate immune systems; it was expected to have the abilities of recognition of known intruders, to learn about previously unknown intruders.

S. Forrest et al. [2, 4] first brought intrusion detection research interest into program behavior data for UNIX privileged processes. Processes/programs are viewed as black boxes; observables can be monitored as the run time characteristics of processes/programs. System calls are good observables because they involve system resources accessed by processes/programs. The authors also introduced the "Self" and "Non self" concepts from immune systems. In an immune system, discrimination is based on a characteristic structure called peptide, a short protein fragment. So they proposed the N-grams method, where a profile of normal behavior is built by enumerating all unique, contiguous, length-N sequences that occur in training data. In the classification process, unmatched sequences are labeled as anomalous. Forrest et al brought anomaly detection into a new realm. Because data from system calls display sequential and ordered behavior, this data contains rich information for anomaly detection.

Later on, researchers extended the N-grams approach by applying machine learning algorithms, such as neural networks or rule-based classifiers. C. Warrender et al. [8] used Hidden Markov Model (HMM) to model sequential system call traces. They took advantage of HMM to classify the entire trace instead of dividing the trace into N-grams, so they could capture the long range dependencies in traces. C. Zou [10] tried HMM to model the user's behavior. The results were not good for the following reasons: (1) user behavior data that he used was collected from a company giving too little information to distinguish between normal and anomaly behaviors. (2) Anomaly intrusion detection is a very hard topic. By now, it is still primarily an academic research area in its infancy. (3) HMM is suitable for sequence classification in one-dimensional data, like voice wave or spectrum. Some anomaly detection data are multi-dimensional sequences with continuous and discrete variables intertwined. In such cases, it seems that using HMM alone is not quite suitable for the anomaly intrusion detection task. N. Ye et al. [9] used first order and high order stochastic models to classify sequential audit event data. They captured the transition of events for intrusion detection. They calculated the event transition probabilities and evaluated the joint probability of sequences to classify intrusive events.

3 Data Collection

3.1 Malicious Codes and Data Collection

1) Malicious codes

A comprehensive and well classified data set of malicious codes was integral to our study of analyzing and detecting malicious binaries in Microsoft Windows systems. We were greatly helped by the availability of such data on the Internet. Several categories of malicious code were collected from a mirror website of VX Heavens [3]. The current malicious codes are typically named as follows:

TopLevelCategory.SubCategory.MalwareName.Variant

For example, backdoor.win32.darkftp.14 is a malicious file. The Top level category is backdoor; the Sub-Category field is win32; the malicious name field is darkftp; the Variant field is 14. The Top level category field describes the type of malicious code and is useful in determining the intended effect of the malicious code. For example, Email-flooder would send out a large number of emails. A Backdoor would most likely provide a secret open-port or an access authentication which only the attacker is aware of. The Sub-Category field represents the target program or platform which the malicious code is attacking or exploiting. The malicious name field is what that program is most popularly known as on the Internet. The Variant field is usually a small modification to the original malicious code and is represented with either a number or a letter in the alphabet depending on the number of variants available.

Table 1: Count of malicious codes by file type

Malicious codes by target platform or application	Count of malicious codes
BAT	1521
Java	60
VBS	896
IIS	55
MS Excel	311
MS Word	2723
DOS	18733
Win32	10135
Win16	484
Linux	188
Unix	95

We focus here only on Win32 malicious executables. Table 1 shows the distribution of malicious codes based on their file type. DOS malicious codes are the most frequent; they have been steadily on the decline while the number of Win32 malicious codes has risen sharply in recent years. The Win32 platform is currently the dominant operating system in the personal computer arena. These are the reasons for focusing exclusively on the Win32 file types in this study.

As intrusion detection techniques evolve, the Windows operating system evolves as well. We have already experienced the DOS, Win16, and Win32 operating systems, and the next generation will be a Win64 platform. In this study, the benign executables, malicious executables and data collection components are related to the Win32 platform. However, the profile generation and detection models are platform independent. So we can apply our approach to other platforms without significant modification.

Malicious codes can also be classified by the file infection techniques that they implement. Each file infection technique specifies what malicious activity is to be performed. For example, an overwriting malicious code simply locates a critical file and overwrites it completely with its own code. One obvious effect of such viruses is that the system may then be incapable to perform its basic functions. If we knew the file infection techniques used by malicious codes, it could help us understand their program behavior pattern by comparing with their intended activities. But such information is not readily available. Thus, we stick to the first taxonomy of malicious codes.

2) Safe Execution Environment and Malicious Data Collection

One major difficulty of malicious data collection is that we do not know how much damage the system is exposed to when critical executables are run. The bottom line is that we cannot allow the malicious executables infect or crash the physical machine. A safe execution environment is needed for running untrusted programs. Our solution is to build a virtual environment and run the malicious executables on the virtual platform. Therefore, we could contain any potential damage to the physical machine.

The virtual platform was set up by using Microsoft Virtual PC 2004, which is a powerful software virtualization solution that allows us to run multiple PC-based operating systems simultaneously on one workstation, providing a safety net to maintain compatibility with legacy applications while migrating to a new operating system. It also saves reconfiguration time, so support, development, and training assignments can be done more efficiently. The Virtual PC technology serves a variety of purposes. Key applications for Microsoft Virtual PC 2004 include legacy application support, tech support, training, and consolidation of physical computers; it provides a safety net for OS migration, execution of critical programs/applications, and rapid reconfiguration.

The Virtual PC provides a running time environment which is not much different than the one on the real system. In addition, the Virtual PC platform provides a two-way isolation environment from the host operating system. Programs running on the Virtual PC are not given read-access and write-access to the host environment. Read-access and write–access are restricted within the virtual environment. So even under severe consequences of malicious code execution, only the virtual environment will be infected, aborted, or crashed. Under careful operation, we fully contain any damages caused by malicious executables inside the virtual platform without affecting the host machine. Similarly, W. Sun et al. [3] proposed a safe execution environment tool called Alkatraz which provides a one-way isolation for the host system. Programs running within the safe execution environment are given read-access to the host system, whilst write access is restricted within the safe execution environment.

3.2 Data Types

In contrast to static analysis, dynamic analysis involves the actual execution of a file to collect run-time data. This technique has certain advantages over static analysis, such as the availability of de-obfuscated data, real-time execution sequence data, actual values for operating system call parameters, etc. In our approach we collect data for the following:

- System calls.
- Registry accesses.
- File system accesses.

System call data provide comprehensive information on system call sequences. Registry access data contain information for registry calls made by a program/process at run-time. File system access data contain information for file system calls made by a program/process at run-time. Both of the last two types of data are a special subset of system calls.

A system call is a software interrupt used by an application program to request service from the operating system. System calls often use a special machine code instruction which causes the processor to change mode or context (e.g. from the "user mode" to the "protected mode"). As with the Hexdump utility, a tool called 'Strace' is available on UNIX or Windows operating systems; it is a popular tool used to print the trace of system calls made by a process/program.

4 Forward Detection Model

4.1 Introduction

In anomaly detection systems, the critical component is normal profile generation. The normal profile forms the foundation for anomaly detection. A poorly established profile cannot serve reliable prediction and detection of malicious attacks. A good profile includes proper data, feature selection, physical logic, and a mathematical model. We will briefly explain these components. Our preliminary investigation found out that registry access data and file system access data are basically subsets of the system call trace data. So we focus on system call traces to build profiles. System call trace data look like:

1 133 139 NtOpenKey (0x80000000, {24, 0, 0x40, 0, 0, "\Registry\Machine [...]

2 133 139 NtCreateEvent (0x100003, 0x0, 1, 0, ... (0x71951000), 4096, 4,)) == 0x0

3 133 139 NtAllocateVirtualMemory (-1, 1243984, 0, 1244028, 8192, 4, ...) == 0x0

4 133 139 NtAllocateVirtualMemory (-1, 1243980, 0, 1244032, 4096, 4, ...) == 0x0

5 133 139 NtAllocateVirtualMemory (-1, 1243584, 0, 1243644, 4096, 4, ...) == 0x0

......

The first column is an identity, which lets you match up calls that do not complete immediately (and are broken into two lines). The second and third columns are the IDs of the process and thread making the call. Next is the name of the system call, the input parameters, three dots (...), then output parameters, and the return code. The useful information is the system call, input parameter and output parameter. However, we will only use the system call information. First, the input parameters and output parameters could be highly environment dependent. For example, one system call is:

NtOpenFile (0x100020, {24, 0, 0x40, 0, 0, "C:\Program Files\MATLAB\Man.Manifest"}, 5, 96, ... 56 == 0x0

Even under the same operating system, the path parameter could be varied. What we want is a profile that is independent of any particular system or application environment. If we stick to specific parameters related to individual programs, then we will have trouble identifying the general patterns of normal and malicious programs. This, in turn, will defeat our purpose of anomaly detection for new threats. Also, there may be millions of input and output parameters that could require extraordinary computing resources, if they were to be accounted for. Windows 32 systems have more frequently used system calls than UNIX systems, but their total number is still in the hundreds.

By carefully investigating the data, we also found some "physical reasoning" for anomaly detection. Normal programs used by ordinary users are usually well written (by software vendors), whereas malicious programs are usually written by individuals and usually have more errors. For example, normal programs usually call multiple objects, whereas malicious programs usually call only one object; malicious programs also raise more exceptions than normal programs. From our collected statistics, we found out that "NtWaitForMultipleObjects" has higher frequency in normal program behavior, and "NtWaitForSingleObjects" has higher frequency in malicious program behavior. A hacker trying to access a computer system is similar to a thief trying to break into a house, whilst a legal user trying to access a computer system is similar to the owner entering his own house. Thieves usually are not familiar with the environment, so malicious programs will query more system and user information. Malicious programs also have more registry accesses, file system accesses, backdoor attempts and other anomalous attempts than normal ones. Table 2 shows data for some system calls that support the above reasoning.

System call trace data is sequential data. We can view the system call trace as a discrete-time stochastic process. A discrete-time stochastic process is a sequence $X_1, X_2, X_3, \ldots$ of random variables. The range of these variables, i.e. the set of their possible values, is called the state space. The value of X_i is the state of the process at time i. The conditional probability distribution of X_{i+1} on past states is shown as the function: $Pr(X_{i+1} = x | X_1, X_2, X_3, \ldots, X_i)$.

Table 2: Some system call counts for normal and malicious programs

System call name	Count in all normal programs	Count in malicious programs
NtWaitForMultipleObjects	101317	6571
NtWaitForSingleObject	14872	118724
NtQueryVirtualMemory	17497	3681
NtDelayExecution	14503	234143
NtRaiseException	519	27911
NtUserSystemParametersInfo	581	7030
NtQuerySystemInformation	6235	22355
...	...	...
Total number of system calls	1028600	2507000

In most circumstances, the conditional probability distribution of X_{i+1} on past states is a function on X_i alone:

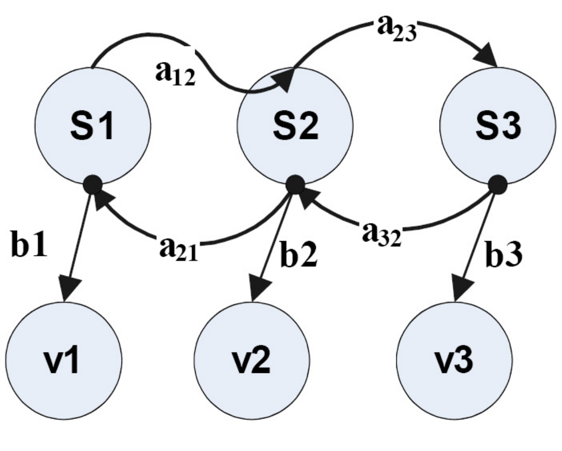
$$Pr(X_{i+1} = x | X_1, X_2, X_3, \ldots, X_i) = Pr(X_{i+1} = x | X_i)$$

The identity above identifies a Markov chain, and the conditional distribution is called the transition probability of the process. The Markov chain serves as a powerful mathematical tool to deal with a random process; it has many applications in physics, biology, statistics, geostatistics, etc. Since Forrest et al. used N-grams for sequential system call trace data, most of the researchers have explicitly or implicitly assumed Markov chains or state transitions in program behavior analysis. Basically, they built the normal profile by explicitly or implicitly computing the state transition probabilities, then evaluated the conditional probability of an incoming instance (trace or segments) based on the built profile. We name such attempts forward models. We implemented forward models in our solution.

We extended "Self" into a more general concept. In our context, the family of all good cells (all normal program behaviors) are defined as "Self", while all other cells (behaviors) falling out of the boundary of "Self" are defined as "Non-self". Instead of repeatedly collecting identical process/program data, we collected data from different, unique programs. We collected data from 493 unique, normal Windows programs and 525 unique, malicious Windows programs. Basically, we built a normal profile based on system call traces collected from different normal programs, and used this profile to detect anomalies due to malicious program behavior.

4.2 Hidden Markov Model

HMM is a doubly stochastic process that has been widely used in speech recognition and DNA sequence modeling. Each HMM contains a finite number of observable and hidden states. From the normal data, we train the HMM to maximize the likelihood of training sequences. Then we use the trained HMM to calculate the likeli-



S_i — States of the Markov model.
a_{ij} — Transition probabilities.
b_j — Output probabilities.
v_i — Observable outputs.

Figure 1: An illustration of the hidden Markov model

hood (log-likelihood) of testing traces. If the likelihood (log-likelihood) is low, it is likely to be a malicious trace; otherwise, it is probably a normal trace.

In the Hidden Markov Model [5], a sequence is modeled as $\lambda = (A, B, \pi)$ where:

$A = \{a_{ij}\}$ is the state transition probability from state S_i to state S_j, $a_{ij} = P(X_{t+1} = S_j | X_t = S_i), 1 \leq i, j \leq N$;

$S = \{S_1, S_2, ..., S_N\}$ is the finite set of possible states;

$B = \{b_j(k)\}$ is the set of observation symbol probabilities at state S_j of time t, $b_j(k) = P(v_k | X_t = S_j), 1 \leq j \leq N, 1 \leq k \leq M$;

$V = \{v_1, v_2, ..., v_M\}$ is the finite set of possible observations;

$\pi = \{\pi_i\}$ is the initial state distribution, $\pi_i = P(X_i = S_i), 1 \leq i \leq N$.

HMM learning can be conducted by the Baum-Welch or forward-backward algorithm [5], an example of a generalized Expectation-Maximization algorithm. An estimation of parameters $\lambda = (\overline{A}, \overline{B}, \overline{\pi})$ is first initialed. After the learning process is applied with the training set, the normal profile can be modeled with the HMM of $\lambda = (A, B, \pi)$.

In our problem, the normal traces are basically different from each other, so "Self" is difficult to learn from normal traces, because "Self" needs some repetitions to be reinforced. So we broke the normal traces into N-grams; this way we get some repetitions from the list of N-grams. Also, by utilizing N-grams, we do not need to consider the varied length of the training traces.

The standard HMM requires a fixed number of states, so we must decide on their number before training. Classic theory shows that a good choice for the application is to choose a number of states roughly corresponding to the number of observation states, i.e. the number of unique system calls used by the programs. Therefore, choosing the number of states depends on the dataset used in the experiments. The Windows OS has more system calls than the UNIX OS. And we model hundreds of different normal program traces, instead of one program/process trace. In the original data, we have more than 400 observable states. The dimensionality is too large. It will cause problems in training and increase the computational cost. Previous works have used 20-60 observable states. So we selected some system calls (around 30) based on the frequency of appearance and difference in frequency between normal and malicious traces; we combine all the other low frequency system calls into one observable state. The reasons are: (1) if we consider these very low frequency system calls, they will bring certain oscillation into the performance of the model; (2) If there is an unseen observable state in the evaluation data, it will be very difficult for the HMM to handle; this situation is pretty normal in malicious activities; (3) finally, we have to consider the computation cost, even if we lose slightly on accuracy. Most of the time, a trade off is required for a good balance.

Given a testing system call trace, most efforts use a sliding window of length N to move along the trace to get short sequences; then they evaluate those N-grams. By this process, they avoid considering the effect of varied length of traces. However, we modified the evaluation and the detection components. Instead of short sequences, we evaluate the whole sequence of each trace. By this, we can learn some overall characteristics for each trace. But traces are of different length. As we know, generally the longer the trace, the less the likelihood (or probability) of this trace, based on the learning HMM. So we identified a relationship between the length and the likelihood of traces.

The evaluation is based on the normal profile $\lambda = (A, B, \pi)$ from the learning process. The likelihood $P(OS|\lambda)$ (or probability) of a given output observable sequence $OS = \{OS_1, OS_2, OS_3, ..., OS_N\}$ was evaluated by the forward algorithm. For scaling purposes, instead of the likelihood, we use log-likelihood $log(P(OS|\lambda))$ of the sequence.

We used the proposition that there is roughly a linear relationship between the length and the log-likelihood of normal traces. We give a simple proof here. Let's say we have two homogeneous sequences: $X = \{X_1, X_2, ..., X_N\}$, and $Y = \{Y_1, Y_2, ..., Y_M\}$, with $\frac{log(P(X|\lambda))}{Length(X)} \approx k$, $\frac{log(P(Y|\lambda))}{Length(Y)} \approx k$, and k is a constant, then for the combined sequence $Z = \{X_1, X_2, ..., X_N, Y_1, Y_2, ..., Y_M\}$, we

Table 3: Classification result (experiment 1): number of states = 30, number of observations = 31

	525 abnormal files	156 normal files
Correctly classified (rate)	456 (86.9%)	139 (89.1%)
Misclassified (rate)	69 (13.1%)	17 (10.9%)

have

$$\begin{aligned} & log(P(Z|\lambda)) \\ = \; & log(P(X|\lambda) * P(Y_1|X_N) * P(Y|\lambda)) \\ = \; & log(P(X|\lambda)) + log(P(Y_1|X_N)) + log(P(Y|\lambda)). \end{aligned}$$

$$\begin{aligned} & \frac{log(P(Z|\lambda))}{Length(Z)} \\ = \; & \frac{log(P(X|\lambda)) + log(P(Y_1|X_N)) + log(P(Y|\lambda))}{N+M} \\ \approx \; & \frac{k * N + O(1) + k * M}{N+M} \\ = \; & k + \frac{O(1)}{N+M} \approx k. \end{aligned}$$

This could be viewed as a special case of the large number theorem. A large data set (here a long sequence) will show some average statistical pattern. This conclusion was also empirically verified by the data we collected. In a testing data set of 101 normal traces, the correlation coefficient of their length and their log-likelihood (based on the normal profile) is -0.996. It is quite close to -1, which means that k is a negative constant. The normal traces share some homogeneous characteristics. In a testing data set of 525 malicious traces, the correlation coefficient of their length and their log-likelihood (based on the normal profile) is -0.876, which means that they are not so homogeneous.

Once the normal profile was built, it is obvious that normal traces are more similar to the normal profile than malicious traces. In general, normal traces may require likely transitions, and malicious traces may need more unusual transitions. So normal traces will give higher log-likelihood (likelihood) values, and malicious ones will give lower log-likelihood (likelihood) values relatively to their length. So we define the anomaly index for trace T as:

$$AI(T) = -\frac{log(likelihood(T))}{Length(T)}.$$

The negative sign just makes the anomaly index positive. A threshold was determined. If the anomaly index was larger than the threshold, then the trace will be considered as anomalous.

Table 4: Classification result (experiment 2): number of states = 20, number of observations = 33

	525 abnormal files	156 normal files
Correctly classified (rate)	327 (62.3%)	136 (87.2%)
Misclassified (rate)	198 (37.7%)	20 (12.8%)

Table 5: Classification result (experiment 3): number of states = 33, number of observations = 33

	525 abnormal files	156 normal files
Correctly classified (rate)	438 (83.4%)	136 (87.2%)
Misclassified (rate)	87 (16.6%)	20 (12.8%)

4.3 Experimental Results

Some experimental results are shown in Tables 3, 4, 5. From the experiments we found out that when the number of states is close to the number of observations in HMM, a better classification result will be derived, which has already been asserted by others. If we focus on experiments 1 and 3, we observe 85-88% overall accuracy on all testing traces, and 83-87% accuracy on testing malicious traces.

5 Conclusion

We presented an anomaly detection model based on program behavior in the Windows environment. To the best of our knowledge, this is the first effort focusing on dynamic analysis. We built a profile for normal behavior based on Hidden Markov Model. The results are very promising. Our analysis could benefit from additional real world data. In the near future, we hope that we can collect more real scenario data to implement our framework in real time, and also to improve its execution performance.

References

[1] D. E. Dening, "An intrusion detection model," *Proceedings of IEEE Symposium on Security and Privacy*, pp. 118–133, 1986.

[2] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," *Proceedings IEEE Symposium on Computer Security and Privacy*, pp. 120–128, 1996.

[3] VX Heavens, accessed May 11, 2007. (http://vx.netlux.org/)

[4] S. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of Computer Security*, vol. 6, pp. 151-180, 1998.

[5] K. Jain, and R. Sekar, "User-level infrastructure for system call interposition: A platform for intrusion detection and confinement," *Proceedings of Network and Distributed Systems Security Symposium*, 2000.

[6] J. O. Kephart, "A biologically inspired immune system for computers," *Artificial Life IV: Proceedings of the 4th Int'l Workshop on Synthesis and Simulation of Living Systems*, pp. 130-139, MIT Press, 1994.

[7] R. Wang, *Flash in the pan?*, Virus Bulletin, Virus Analysis Library, July 1998.

[8] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," *Proceedings of IEEE Symposium on Security and Privacy*, pp. 133-145, 1999.

[9] N. Ye, T. Ehiabor, and Y. Zhang, "First-order versus high-order stochastic models for computer cntrusion detection," *Quality and Reliability Engineering International*, vol. 18, pp. 243-250, 2002.

[10] C. Zou, *Using Hidden Markov Model in Anomaly Intrusion Detection*, University of Central Florida, Accessed Oct. 6, 2006. (http://www.cs.ucf.edu/czou /research.htm)

Xin Tang received his B.S. degree in pure mathematics from Peking University, China, in 1992, M.S. degree in applied mathematics from New Jersey Institute of Technology, U.S.A., in 2000, and the Ph.D. degree in computer engineering from New Jersey Institute of Technology in 2007. His research interests include intrusion detection, pattern recognition and program behavior detection.

Constantine M. Manikopoulos received his Ph.D. degree from Princeton University. He was an Associate Professor in the ECE Department at NJIT until 2007.

Sotirios G. Ziavras received the Diploma in EE from the National Technical University of Athens, Greece, in 1984, the M.Sc. in Computer Engineering from Ohio University in 1985, and the D.Sc. degree in Computer Science from George Washington University in 1990. He was a Distinguished Graduate Teaching Assistant at GWU. He was also with the Center for Automation Research at the University of Maryland, College Park, from 1988 to 1989. He is a Professor in the ECE Department at NJIT. He is an author of about 140 research papers. He is listed, among others, in Who's Who in Science and Engineering, Who's Who in America, Who's Who in the World, and Who's Who in the East. His main research interests are computer architecture, reconfigurable computing, embedded computing systems, parallel and distributed computer architectures and algorithms, and network router design. He is a senior member of the IEEE.