A Study on Masquerade Detection

Lin Huang

A Project Report
Presented to
The Faculty of the Department of Computer Science
San Jose State University

submitted in partial fulfillment of requirements for the degree

MASTER OF COMPUTER SCIENCE

at the

SAN JOSE STATE UNIVERSITY

December, 2010

This report has been approved
for the Department of Computer Science
and the College of Graduate Studies by

Dr. Mark Stamp	Computer Science Department	Date

Dr. Robert Chun	Computer Science Department	Date

Dr. Teng Moh	Computer Science Department	Date

ACKNOWLEDGMENTS

I am indebted to my advisor, Dr. Mark Stamp, for his consistent guidance, support, and encouragement throughout my master program. Dr. Mark Stamp has tirelessly guided me on how to perform meaningful research at every step. He has been and will always be an excellent role model for me.

I would like to specially thank Dr. Robert Chun and Dr. Teng Moh for serving on my defense committee.

I also would like to thank my friends, Fan Yang, Leyan Tang, and Yue Chen. They have made my life in San Jose enjoyable and memorable. I am especially grateful to my dear wife Xiuduan Fang for her everlasting love, encouragement and support.

ABSTRACT

In modern computer systems, usernames and passwords have been by far the most common forms of authentication. A security system relying only on password protection is defenseless when the passwords of legitimate users are compromised. A masquerader can impersonate a legitimate user by using a compromised password.

An intrusion detection system (IDS) can provide an additional level of protection for a security system by inspecting user behavior. In terms of detection techniques, there are two types of IDSs: signature-based detection and anomaly-based detection. An anomaly-based intrusion detection technique consists of two steps: 1) creating a normal behavior model for legitimate users during the training process, 2) analyzing user behavior against the model during the detection process.

In this project, we concentrate on masquerade detection, a specific type of anomaly-based IDS. We have first explored suitable techniques to build a normal behavior model for masquerade detection. After studying two existing modeling techniques, N-gram frequency and hidden Markov models (HMMs), we have developed a novel approach based on profile hidden Markov models (PHMMs). Then we have analyzed these three approaches using the classical Schonlau data set. To find the best detection results, we have also conducted sensitivity analysis on the modeling parameters. However, we have found that our proposed PHMMs do not outperform the corresponding HMMs. We conjectured that Schonlau data set lacked the position information required by the PHMMs. To verify this conjecture, we have also generated several data sets with position information. Our experimental results show that when there is no sufficient training data, the PHMMs yield considerably better detection results than the

corresponding HMMs since the generated position information is significantly helpful for the PHMMs.

TABLE OF CONTENTS

Page

NOMENCLATURE

IDS: Intrusion detection system.

Alert/Alarm: A signal suggesting that a system has been or is being attacked [1].

True Positive: A legitimate attack which triggers IDS to produce an alarm [1].

False Positive: An event signaling IDS to produce an alarm when no attack has been taken place [1].

True Negative: When no attack has taken place and no alarm is raised [1].

False Negative: A failure of IDS to detect an actual attack [1].

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

1. Introduction

1.1. Intrusion Detection Systems (IDS)

In modern computer systems, usernames and passwords have been by far the most common forms of authentication. A security system relying only on password protection is defenseless when the passwords of legitimate users are compromised. A masquerader can impersonate a legitimate user by using a compromised password.

To detect this issue of masquerading user, Intrusion Detection Systems (IDSs) have been proposed to provide an additional protection for the system by inspecting user behavior [2]. The basic approach used by an IDS is to monitor ongoing activities within the system and to look for malicious or unusual behaviors. Once the IDS concludes a harmful activity has occurred, further actions can be taken to intervene, such as raising an alarm or blocking the user's session. From the perspective of detection techniques, there are two general detection techniques used by IDSs: signature-based detection (also known as misuse detection) and anomaly-based detection.

1.1.1. Signature-based Detection and Anomaly-based Detection

Signature-based detection systems depend on predetermined patterns that represent misuse. Such a pattern should summarize the distinctive characteristics of an attack, often referred to as the signature of an attack. In the detection phase, the IDS records and inspects user activities, and then looks for events that match a predefined

pattern. If a match is found, the detection system raises an intrusion alarm. As a result, a signature-based system is very accurate for detecting known attacks. Moreover, with the information associated with the signature, the IDS is able to give a concrete description of the threat when raising an alarm [2]. However, a signature-based detection system cannot detect unknown attacks. Without the signatures of new attacks, the IDS knows nothing about such an attack. There is always a lag between the time when a new attack is found in the wild and the integration of its signature into the IDS database. Therefore, it is crucial for the signature database to be continuously updated to include new attacks.

Anomaly-based detection systems model user behavior to determine the characteristics of a user's normal behavior [2]. During the detection phase, anomaly-based systems record and analyze user activities and compare this against their normal behavior model. A deviation from the established behavior model is considered an anomaly, or an indication of a possible attack. Often a threshold value is used to define how much deviation will be required before an anomaly is considered an intrusion. Anomaly-based detection systems can detect both known and unknown attacks, provided that the attacker's behavior is significantly different from that of the normal user.

One major challenge of the anomaly-based approach is to model normal behavior. To construct such a model, we must extract distinct characteristics of user behavior. We should also collect a sufficient amount of user behavior data for training purposes. Of course, the user behavior data must be collected under conditions where no intrusion is in progress [2]. A threshold value is needed to indicate how much deviation will be considered as an intrusion. Selecting a threshold value presents a tradeoff between the false position rate and the false negative rate.

User behavior will almost certainly change over time. Without an updating mechanism, an established behavior model will become obsolete, resulting in a large number of false positives. To overcome this problem, most anomaly-based IDSs will update to a new “normal” so that the model can adapt to changes. While this approach deals with the normal user’s changing behavior problem, it also leaves a potential security loophole. An intruder can cheat an IDS into believing he is a legitimate user by acting like a normal user and only gradually changing his behavior [2].

1.1.2. Performance Criteria

In most anomaly-based IDSs, there is a mechanism to score test data, and a threshold value is set to determine whether a piece of data is more likely from an original user or an intrusion user. For example, given a threshold value, if an input is evaluated to have a score higher than the threshold, this input will be categorized as normal data; otherwise, it will be treated as intrusion data.

The threshold value has a significant opposite effect on the false positive (false alarm) rate and the false negative (miss target) rate. If we increase the threshold to catch more intrusion data, the false negative rate will decline; however, the false positive rate will increase since more normal data will be categorized as intrusion data. Conversely, if we lower the threshold, the false negative rate will increase but the false positive rate will decrease. Therefore, the threshold value presents a tradeoff between the false positive rate and false negative rate, since neither high false positive rate nor high false negative rate is desirable. High false negative rate leaves many intrusions uncaught, making IDSs

useless. High false positive rate, on the other hand, floods IDSs with a large amount of false alarms, eventually causing administrators to ignore true intrusion alarms along with false alarms.

To better compare the performance amongst different intrusion detection techniques, we use the Receiver Operating Characteristics (ROC) curve [10] and scatter charts to analyze masquerade detection results. The ROC chart shows the overall detection results for all users, which is useful to compare false positive rates and false negative rates as the threshold value changes. The scatter charts show the detection results of all individual users.

1.2. Masquerade Detection

In this project, we have studied masquerade detection, a specific case of anomaly-based IDS in the UNIX command environment. Note that we use masquerade detection and intrusion detection interchangeably in this report. A masquerader in computer intrusion detection is a person who uses other's computer account [3]. The fundamental assumption of masquerade detection is that each user has his unique characteristics when invoking command sequences. Hence an intrusion likely occurs when there is a significant difference from a user's previous characteristics.

1.2.1. Architecture of a Masquerade Detection System

Figure 1: shows a general architecture of a masquerade detection system. The essential part is to model user normal behavior. Once such a model is constructed, it is

relatively easy to evaluate test data. A good model must preserve the distinct characteristics of each user but ignore trivial information. In masquerade detection, users' historical commands are collected and stored, and then users' ongoing commands are examined based on their historical data.

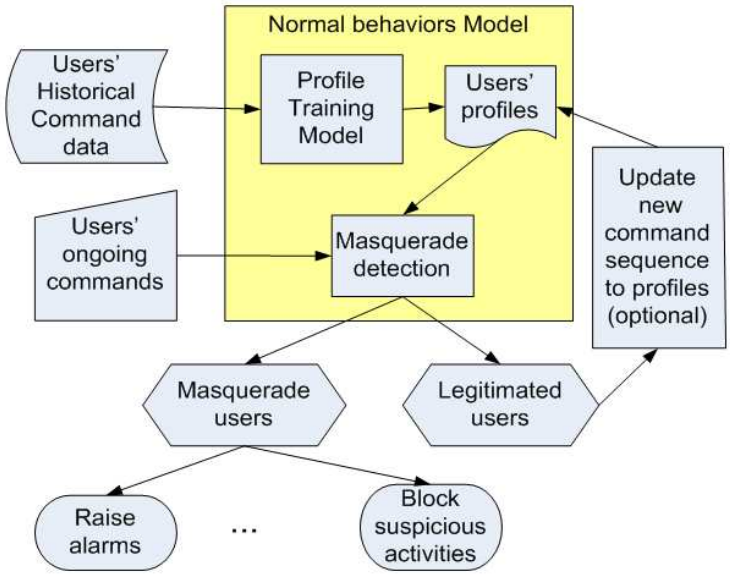

Figure 1: A general architecture of masquerade detection

1.2.2. Schonlau Data Set

Dr. Schonlau has collected masquerading user data for the training and testing purposes for masquerade detection [8]. Figure 2: illustrates the structure of Schonlau data set. This data set consists of 50 data files, one file per user. In each file, there are 15,000 commands (collected using the UNIX audit tool, acct [18]). The first 5000 commands are from an original user and these commands are intended to serve as training data. The

following 10,000 commands are seeded with a masquerader user's commands, and they are intended to serve as test data. The test data can be viewed as 100 command blocks, with 100 commands in each block.

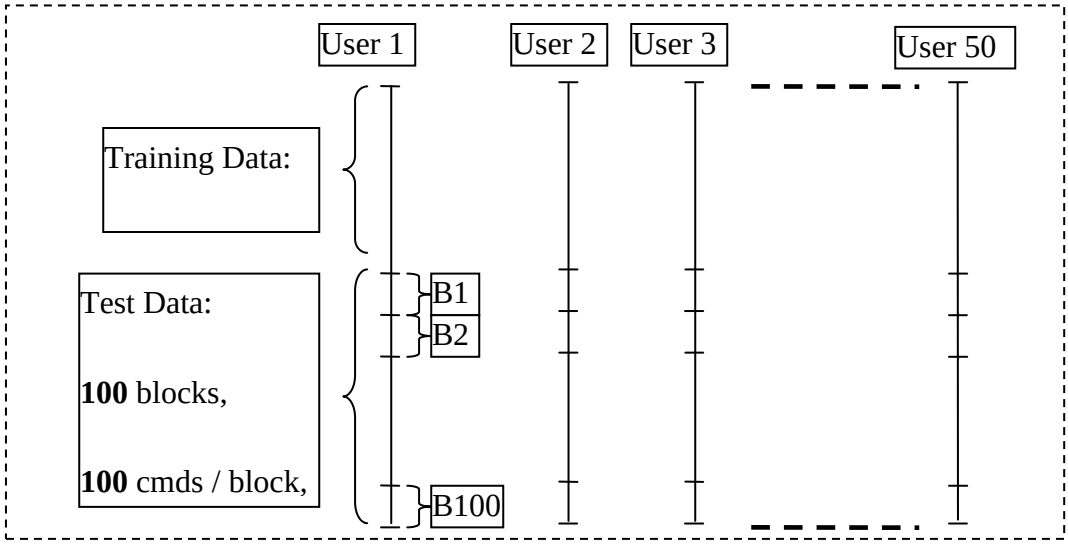

Figure 2: Schonlau Data Set

Schonlau data set contains a map file for the locations of the masqueraded blocks. Figure 3: demonstrates the structure of the map file. The map file contains 100 rows and 50 columns. Each column corresponds to one user, and each row corresponds to a test block. The entries of the map file are set to either 0 or 1. The value of 0 indicates the commands on the corresponding block are not contaminated by a masquerader, and the value of 1 indicates they are contaminated.

The training data provided by Schonlau data set contains only normal behaviors but no intrusion behavior. This is sufficient for masquerade detection since it is a specific case of anomaly-based techniques, which do not require signature of intrusion behaviors.

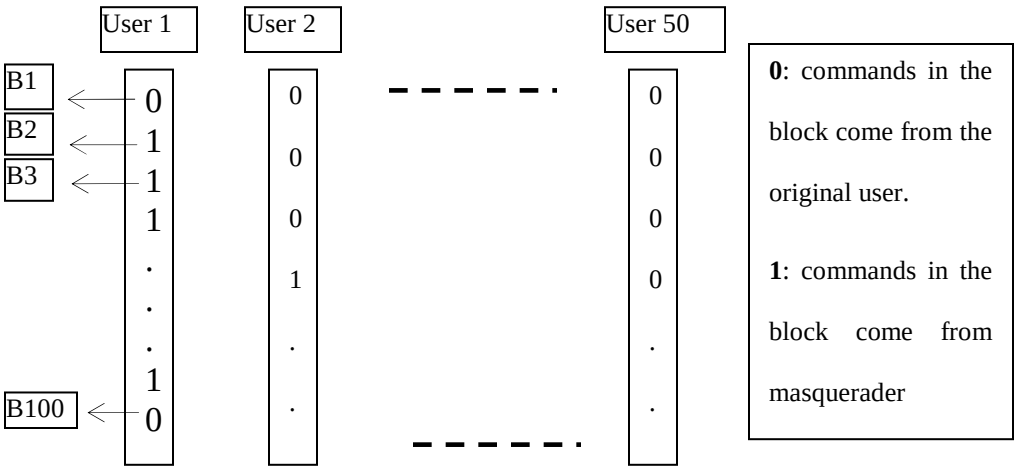

Figure 3: Location of the masquerades

1.3. Project Road Map

This project is focused on different masquerade detection techniques with Schonlau data set as the primary source of training and test data. The goal of this project is to gain an insight on the detection results of different models and to propose a novel model of masquerade detection.

The original project idea was inspired by the similarity between simple substitution cryptanalysis and masquerade detection. Both techniques process observations and try to reveal their underlying truth. In simple substitution cryptanalysis, the observations are cipher texts and the truth is plain texts. In masquerade detection, the observations are test data, and the truth is whether the true identities behind the test data are masquerade users. Simple substitution cryptanalysis has been studied long before the intrusion detection, even before the first computer was invented. Due to the similarity between the two techniques, it is reasonable to apply simple substitution cryptanalysis techniques to masquerade detection. Specifically, we are interested in the normal

behavior modeling techniques and the evaluation functions used by simple substitution cryptanalysis. In Section 2, we present an overview of simple substitution cryptanalysis.

We have reviewed literatures to learn techniques used in simple substitution cryptanalysis, including n-gram frequencies [13,14], double letter, short word patterns, observing syntactic and semantic, relaxation algorithms [19], hidden Markov models (HMMs), genetic algorithms, and dictionary [20,7]. We have analyzed the feasibility of applying these techniques to masquerade detection and found that not all these techniques can be applied. For example, double letter pattern is very useful during word guessing, but the same pattern is not commonly seen in user command sequences. Language semantic is also used to attack simple substitution ciphers. However, it is hard to apply such information to masquerade detection since there are no apparent corresponding semantic for user command sequences. In this project, we have used n-gram frequencies and HMMs to solve masquerade detection.

N-gram frequency statistics is the most fundamental technique to simple substitution cryptanalysis. It has been applied to masquerade detection by treating a command in masquerade detection as a letter in the simple substitution cipher [13, 14]. Intuitively, if a user uses one command frequently now, it is likely for the user to use this command in the near future. Furthermore, if a user executes a group of commands in a certain order, it is also likely for the user to remain this behavior pattern. Thus, it is reasonable to construct a model using the current behaviors and to detect whether the future behaviors fit the trained model. Section 3 describes our work on n-gram frequency statistics for masquerade detection.

HMMs are widely used to uncover hidden states by analyzing a sequence of observations in many areas, such as speech recognition, machine translation, and cryptanalysis. Two HMMs have been studied on Schonlau data set [8] but no sensitive analysis was presented on the key parameters of the HMMs. We have implemented our own configurable HMM and conducted sensitive analysis on the parameters such as the number of states. Section 4 provides a detailed description of applying HMMs to masquerade detection.

An important goal of this project is to design a novel approach for masquerade detection. To our knowledge, there was no study on using profile hidden Markov models (PHMMs) for masquerade detection. PHMMs are commonly used in bioinformatics to effectively find out whether protein sequences are closely related. Unlike HMMs, PHMMs make an explicit use of position information [5]. In the context of masquerade detection, the position represents the order in which a user performs tasks. If a user usually performs tasks in a certain order, PHMMs may be able to take advantage of this position information. Thus, it is reasonable to believe that PHMMs may perform well on masquerade detection. We have designed and implemented PHMMs for masquerade detection, and conducted analysis on experimental results (see Section 5).

After analyzing the detection results, we have found that the PHMMs do not perform as well as the HMMs. We conjectured that it was due to the fact that Schonlau data set lacks session starting and ending information required by the PHMMs. Therefore, we have designed and implemented a model to generate user data with position information. We have found that when there is no sufficient training data, the

PHMMs considerably outperform the corresponding HMMs since the generated position information is significantly helpful for the PHMMs.

2. Simple Substitution Cryptanalysis

2.1. Simple Substitution Cipher

Simple substitution cipher is one of the oldest cipher systems. In a simple substitution cipher, each letter of the plaintext is substituted by another letter. Usually, there is a one-to-one mapping between the letters in the plaintext and the ciphertext. TABLE I: shows an example of simple substitution letter mapping, where the plaintext letters are represented in lower case and the ciphertext letters in upper case, following the convention [2].

TABLE I: AN EXAMPLE OF SIMPLE SUBSTITUTION LETTER MAPPING

Plaintext	a	b	C	d	e	f	g	H	i	J	k	l	m	n	O	p	q	r	s	t	u	v	W	X	y	z
ciphertext	Z	P	B	Y	J	R	G	K	F	L	X	Q	N	W	V	D	H	M	C	U	T	O	I	A	E	S

Using this letter mapping, the plaintext message of `simplesubstitutioncipher` is encrypted into the ciphertext of `CFNDQJCTPCUFUTUFVWBFDKJM` by substituting each letter in the plaintext row with the corresponding letter in the ciphertext row.

To decode the ciphertext message, we can reverse the process by replacing each letter in the ciphertext row with the corresponding letter in the plaintext. For example,

the ciphertext message of `ZWVNZQEPZCJYFWUMTCFVWYJUJBUFVW` will be deciphered as `anomalybasedintrusiondetection`.

2.2. Breaking Simple Substitution Ciphers

As shown in TABLE I:, every permutation of the 26 letters can serve as a simple substitution key. Therefore, the simple substitution has a large key space of 26! ($\sim 2^{88.4}$) since there are 26! permutations in total. However, even with such a huge key space, simple substitution is not sufficiently secure. It can be relatively easy to manually break such a ciphertext by analyzing the letter frequencies and guessing the common words [2]. For example, the attacker can use English letter frequencies as shown in Figure 4: [2]. The nine most frequent letters in English are E, T, A, O, I, N, S, H, and R. After calculating and sorting the letter frequencies in the ciphertext message, an attacker can come up with pretty good guessing by substituting the most frequent letter in the ciphertext with "E", the second most frequent letter with "T", and so on. This approach provides a good start point even if the letter frequencies in the cipthertext may not exactly match the English letter frequencies. In addition, an attacker can adjust the mapping by analyzing the pattern of the letters to guess some common words. For example, *happy* and *hello* have the same letter pattern of ABCCD.

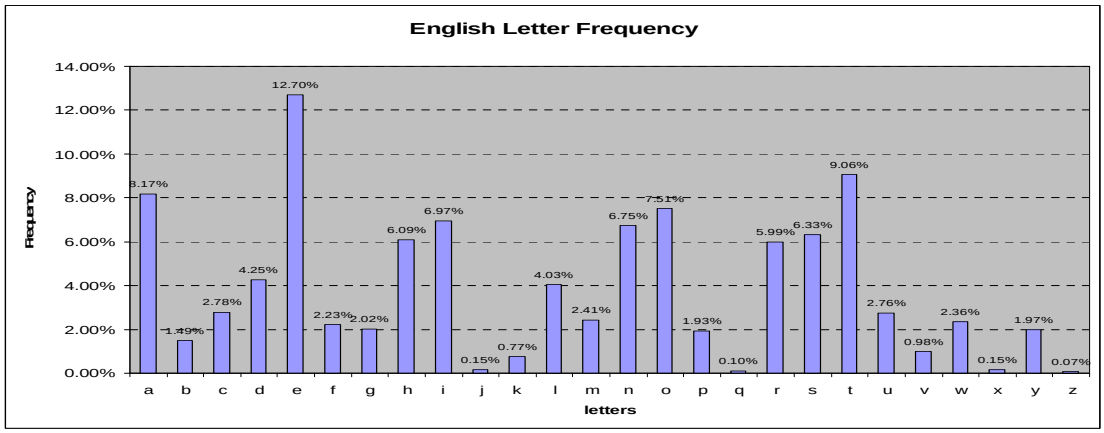

Figure 4: English Letter Frequencies

The above process is a manual schema to break simple substitution ciphers. This manual process requires an attacker to have some knowledge of English to evaluate how sensible a half-broken ciphertext is. To automate the breaking process, a grading method is needed for such an evaluation. To accomplish this task, a decipher system can use much statistics information of the English language, such as the letter frequency counts, bigram frequencies, the most frequent used words, and English grammars. If the grading method is efficient, the decipher system can gradually adjust the key mapping to improve the score of the intermediate deciphered text. Eventually, the decipher system will output a candidate list of plaintexts with high scores. There is a good chance that the original plaintext is amongst the candidate list.

3. Frequency Statistics

3.1. N-gram: Unigram, Bigram, Trigram, and N-gram

Most grading algorithms for simple substitution deciphers are based on N-gram frequencies. An N-gram is a subsequence of n items from a given sequence [13]. An N-gram frequency is the number of the occurrence for an N-gram unit. The 1-gram, 2-gram, and 3-gram are often referred to as unigram, bigram, and trigram, respectively. In the example of Section 2.2, the key mapping guessing is based the English letter frequencies, which is an instant of unigram. Bigram frequency of English letters is used in [6]. TABLE II: shows various n-gram units generated from the command sequence: "**sh xrdb mkpts env csh csh sh csh kill**".

TABLE II: N-GRAM EXAMPLES FOR A COMMAND SEQUENCE

Command sequence	**sh xrdb mkpts env csh csh sh csh kill**
unigram	**Sh, xrdb, mkpts, env, csh, csh, sh, csh, kill**
bigram	**sh xrdb, xrdb mkpts, mkpts env, env csh, csh csh, csh sh, sh csh, csh kill**
trigram	**sh xrdb mkpts, xrdb mkpts env, mkpts env csh, env csh csh, csh csh sh, csh sh csh, sh csh kill**
4-gram	**sh xrdb mkpts env, xrdb mkpts env csh, mkpts env csh csh, env csh csh sh, csh csh sh csh, csh sh csh kill**

To grade a command sequence by using N-gram frequencies, we need to slice the command sequence into N-gram subsequences. Take trigram for example, the command sequence in TABLE II: will be sliced into the trigram items of “sh xrdb mkpts”, “xrdb mkpts env”, …, and “sh csh kill”.

To compute the grading score, we have constructed two frequency lookup tables: the profile lookup table and the command-sequence lookup table. The profile lookup table stores the frequency counts of users’ profiles (i.e., training data). The command-sequence lookup table contains the frequency counts of the command sequences to be evaluated. We use a simple evaluation function to calculate the score for a command sequence:

$$S = \sum_{i=1}^{k} (f_{ui} - f_{ci})^2 \qquad (3.1)$$

where f_{ci} is the normalized frequency count of the *i*th N-gram item in the command-sequence lookup table, f_{ui} is the normalized frequency count in the profile lookup table

corresponding to the f_{ci}, and k is the total number of items in the command-sequence lookup table.

We have tested the unigram, bigram, and trigram frequencies. Both f_{ci} and f_{ui} are normalized to 100. For example, if the training data for a user contains 5,000 commands, the frequency counts will be divided by 50 to be normalized to 100. In this scoring model, a lower score indicates a higher similarity between the training data and test data.

We have calculated the metrics of false negative rates and false positive rates for 1-gram, 2-gram and 3-gram. The false negative rate measures the percentage of actual intrusion uncaught by the IDS. The false positive rate measures the percentage of normal activities that have been recognized as intrusions. See Section 3.2.1 for the experimental results.

To improve the detection results, we add weights to commands since each command is not equally important to every user. We have measured the command weight from two perspectives: 1) the frequency percentage of a command used by each user, 2) the uniqueness of a command to a user.

In terms of the frequency percentage of a command used by each user, we first count the frequency of a particular command used by all users, and then calculate the percentage usage of that command for each user. If a command is used extensively by one user, we assign a higher weight of the command to that user. Intuitively, we have

$$w_{CiUj} = \frac{F_{CiUj}}{F_{CiG}} \qquad (3.2)$$

where *Ci* represents the *i*th N-gram command sequence in the test data, *Uj* represents the user *j*. w_{CiUj} is the weight of *Ci* for user *j*. F_{CiUj} is the frequency count of *Ci* in the training data of user *j*. F_{CiG} is the frequency count of *Ci* in the training data of all users.

Regarding to the uniqueness of a command to a user, if a command is only used by a particular user, then this command is unique to the user. Thus this command should be granted a higher weight. On the other hand, a command used by all users indicates that this is a general command. Since this command carries few characteristics of the user, it should be granted a lower weight. To calculate the uniqueness, we have

$$w_{Ci} = \frac{M}{m_{Ci}} \tag{3.3}$$

where *Ci* represents the *i*th N-gram command sequence in the test data, w_{Ci} is the uniqueness of *Ci*, M is the number of all users, and m_{Ci} is the number of users who have used *Ci*.

These two weighted equations are the simplest ways to compute the significance of an N-gram to a user. More sophisticated calculations have been presented in [13] and [14]. We have experimented with the weighted schemes defined on equation (3.2) and (3.3) for N-gram detection (see Sections 3.2.2 and 3.2.3 for the detection results).

3.2. Experimental Results

3.2.1. 1-gram, 2-gram, and 3-gram

Figure 5: shows the detection results using 1-gram, 2-gram, and 3-gram frequencies. The x axis are the logarithmic values of false positive rates since we are more interested in the detection performance on the lower end of false positive rates, such as less than 5 percent. For convenience, in this report, we denote the region with false positive rates less than 5 percent as the useful zone. A high false positive rate is not practical even though it may be associated with low false negative rates. From Figure 5:, we see that the false negative rates in the useful zone are too high (greater than 70%) to put these n-grams into any practical use.

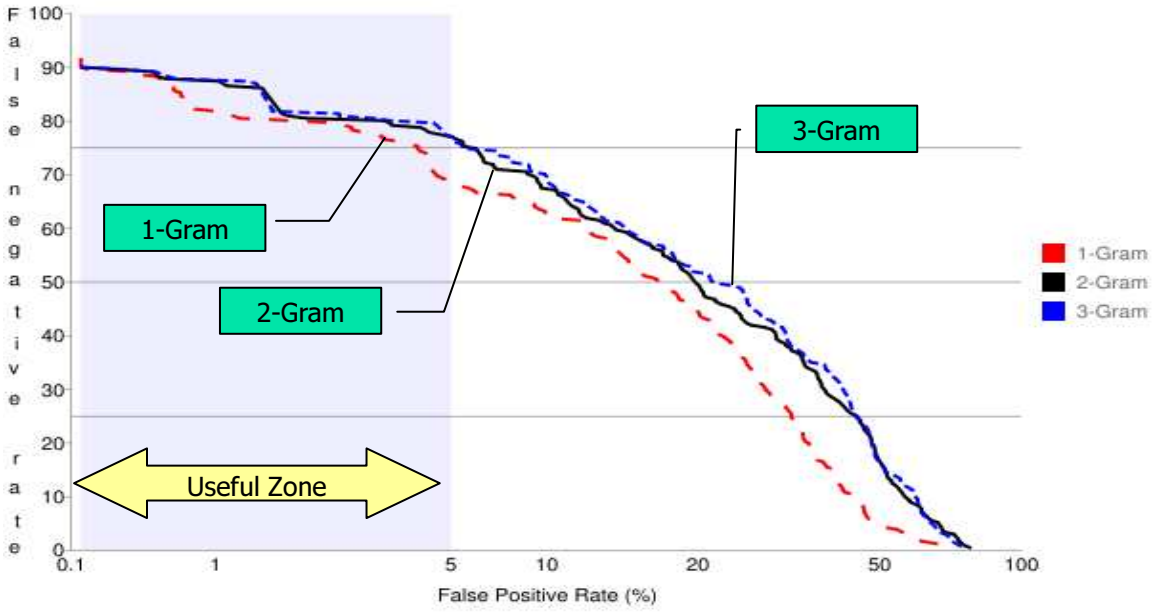

Figure 5: The detection results using 1-gram, 2-gram, and 3-gram

3.2.2. N-gram Weighted by Percentage of Command Usage

Figure 6: shows the detection results of adding the percentage of command usage statistics (defined by (3.2)) to the N-gram models. 1-Gram, 2-Gram, and 3-Gram represent the results of un-weighted N-Gram, while 1-GramCG, 2-GramCG, and 3-GramCG represent the results of weighted N-Gram. From Figure 6:, we see that weighted N-grams significantly outperform the un-weighted versions.

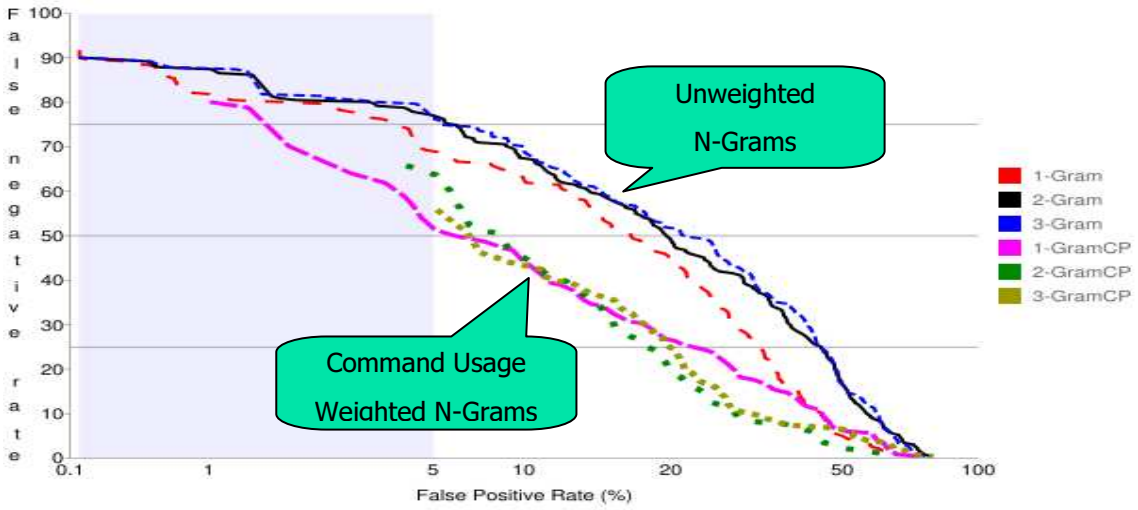

Figure 6: N-grams weighted by percentage of command usage vs. un-weighted N-grams

Note that in the weighted N-Grams, there are some areas with undefined false negative rates when these rates are smaller than some threshold values. For example, the region of false positive rates between 0.1% and 1% is blank for 1-GramCG, and so is the region between 0.1% and 5 % for 3-GramCG. Since we will encounter similar situations

later when we analyze other models, it is worthwhile to investigate why there are such blank areas, and what can be done to eliminate or reduce these areas.

To better understand this problem, let us look at the test data scores from a different perspective. Figure 7: shows the test data scores for User 9. A green diamond represents a normal command block, and a red circle represents a masqueraded command block. The x axis represents the block ids for the 100 test command blocks, and the y axis is the evaluation score for each block. As shown Figure 7:, numerous normal and masqueraded command blocks have been evaluated to the same minimum score of 50, which eventually results in blank areas. The detailed reason goes as follows. During the process of analyzing false positive rates and false negative rates, we gradually change the threshold values to compute these two rates. If we set the threshold value to the minimum score, the IDS will treat every command block as normal data, and therefore, the false negative rate will be 100%. Note that 100% false negative rate is discarded since it carries no useful information. When we slightly increase the threshold value, all these blocks with the same minimum score will be excluded, leading to a dramatic change of false positive rates and false negative rates and resulting in the blank areas as shown in Figure 6:.

We have proposed a fine-tuning approach to eliminate or reduce the blank areas. Specifically, we fine tune the evaluation function so that different command blocks will be evaluated to different score values. Unique values can avoid the dramatic change of false positive rates and false negative rates when threshold is adjusted. The drawback of this method is that there is no universal solution for fine tuning and thus it is time consuming to perform such a task.

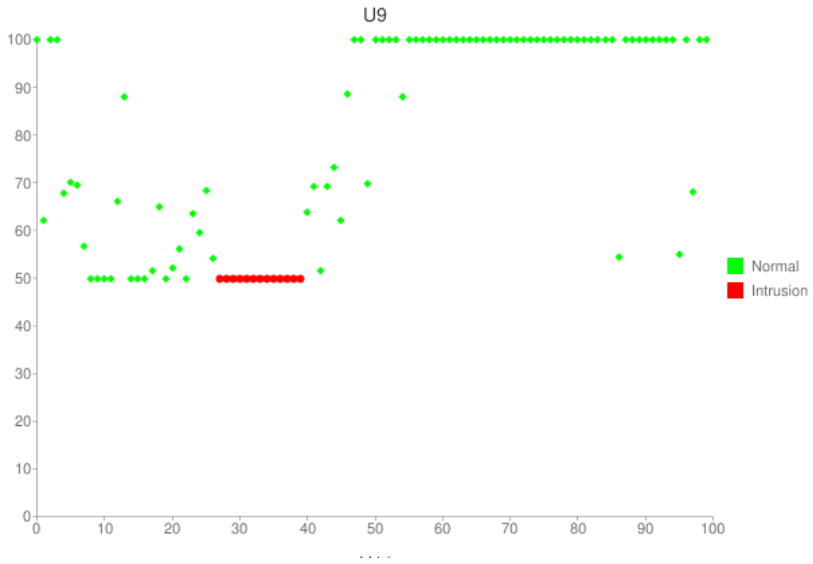

Figure 7: An example of detection results for User 9

3.2.3. N-gram with User Uniqueness

Figure 8: shows the results of adding the uniqueness (defined by (3.3)) to the N-gram. 1-Gram and 3-Gram represent the results of un-weighted N-Gram; 1-GramCG and 3-GramCG represent the results of command usage weighted N-Gram; and 1-GramTF, 3-GramTF represent the results of uniqueness weighted N-Gram. From Figure 8:, we see that the detection results of the uniqueness weighted N-Gram outperform those of the two previous methods.

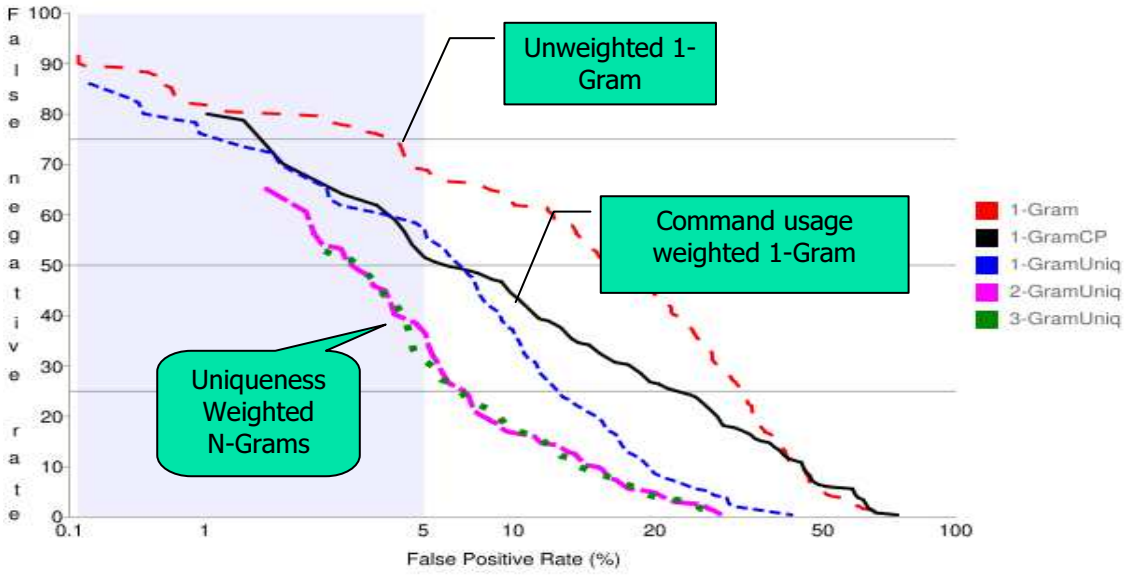

Figure 8: N-gram weighted by uniqueness vs. N-gram weighted by command usage vs. un-weighted N-gram

3.2.4. Conclusions

We have implemented the N-gram models for masquerade detection. We have also analyzed the effects of weighting two factors: 1) the frequency percentage of a command used by each user, 2) the uniqueness of a command to a user. The experimental results show that the false negative rates for weighted N-gram drop significantly while the false positive rates are comparable to those for the un-weighted versions. In particular, the uniqueness weighted N-Gram yields the best performance.

4. HMM: Introduction

4.1. Markov Chain

A Markov chain is a random process of generating a sequence of states using state transition statistics. In a classical Markov chain, the property of the next state depends only on the current state. This model is also known as a first order Markov chain. There are also higher order Markov chains, in which the property of the next state depends not only on the current state but also on previous states. In this report, we are focused on first order Markov chains.

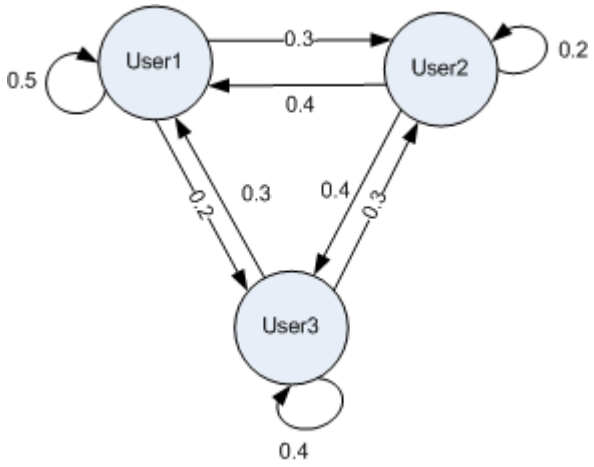

Figure 9: A Markov chain of a computer sharing pattern

Figure 9: shows an example of a Markov chain for a computer sharing pattern. It is assumed that there is only one computer available and this computer is shared by three users, User1, User2, and User3. The usage of this computer is slotted into 15 minutes per unit. Once a user gets the right to use the computer, she/he can use it exclusively for a slot of 15 minutes. When the current slot times out, a new user will be selected for the next time slot. We call one unit time being used by a user a state, which is represented as

a circle in the graph. Thus, there are three states in the system, determined by the user (i.e., User1, User2, and User3).

Assume that we observe the pattern how the users share the computer. For example, if User1 is using the computer in the current timeslot, the probability of User2 will use the computer on the next timeslot is 30%. This relationship is represented as an arrow from User1 to User2, and the probability value 0.3 is associated with the arrow in Figure 9:. In this example, the transition probability from the current state to the next state depends only on the current state, regardless of the previous states. The transition probability of a Markov chain is formally defined as [5]:

$$a_{x_n x_{n+1}} = P(X_{n+1} = x \mid X_1 = x_1, X_2 = x_2, ..., X_n = x_n) = P(X_{n+1} = x \mid X_n = x_n) \qquad (4.1)$$

where $x_i \in$ a countable state set S and X_i is the *i*th observed state in a Markov chain.

An N*N state transition matrix, denoted as *A*, is used to describe the transition probabilities amongst all states, where N is the number of states. For example, the state transition matrix *A* for Figure 9: is

$$A = \begin{bmatrix} 0.5 & 0.3 & 0.2 \\ 0.4 & 0.2 & 0.4 \\ 0.3 & 0.3 & 0.4 \end{bmatrix}$$

Consider the probability for a given state sequence $x_n, x_{n-1}, ..., x_1$ on a specified Markov chain. This probability is determined by the states and the associated state transition probability matrix [5]:

$$P(x) = P(x_n, x_{n-1}, ..., x_1)$$

$$= P(x_n \mid x_{n-1}, ..., x_1)\, P(x_{n-1} \mid x_{n-2}, ..., x_1) \ldots P(x_1)$$

$$= P(x_n \mid x_{n-1}) P(x_{n-1} \mid x_{n-2}) \ldots P(x_2 \mid x_1) P(x_1)$$

$$=P(x_1)\prod_{i=2}^{n} a_{x_{i-1}x_i} \quad (4.2)$$

The “start” and “end” states can be added to a Markov chain to model both ends of an observation sequence. Figure 10: shows such an example.

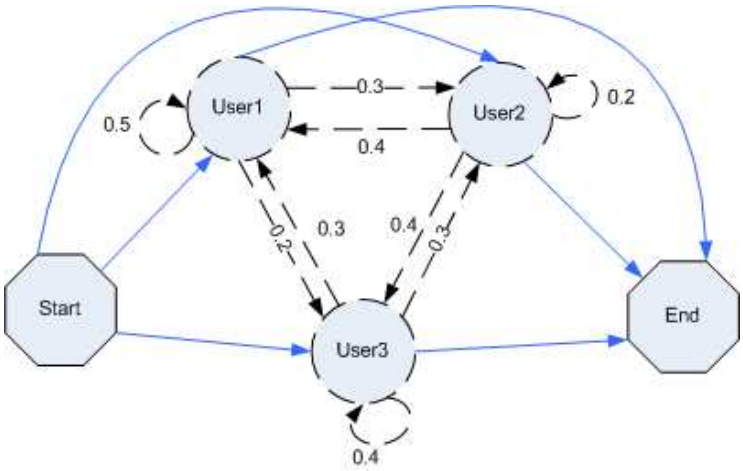

Figure 10: Start and End States are added to the Markov Chain.

4.2. Hidden Markov Model (HMM)

In a regular Markov Model, states are directly visible to the observer. However, in an HMM, states are not directly visible. Instead, only the output, dependent on the states, is visible [11].

To demonstrate a HMM, we modify the previous example. Suppose we would like to track if a user, say User1, is using the computer during a period of time. Assume that the users remotely log in to the computer and we cannot be sure who is using the computer (e.g., the user id may be compromised). To track the usage history of User1, we only consider two states, “User1” and “not User1” as shown in Figure 11:. The “User1”

state means that User1 is using the computer, and "not User1" means another user is using the computer. The state transition, denoted by *A*, can be summarized as:

$$A = \begin{bmatrix} 0.2 & 0.8 \\ 0.4 & 0.6 \end{bmatrix} \tag{4.3}$$

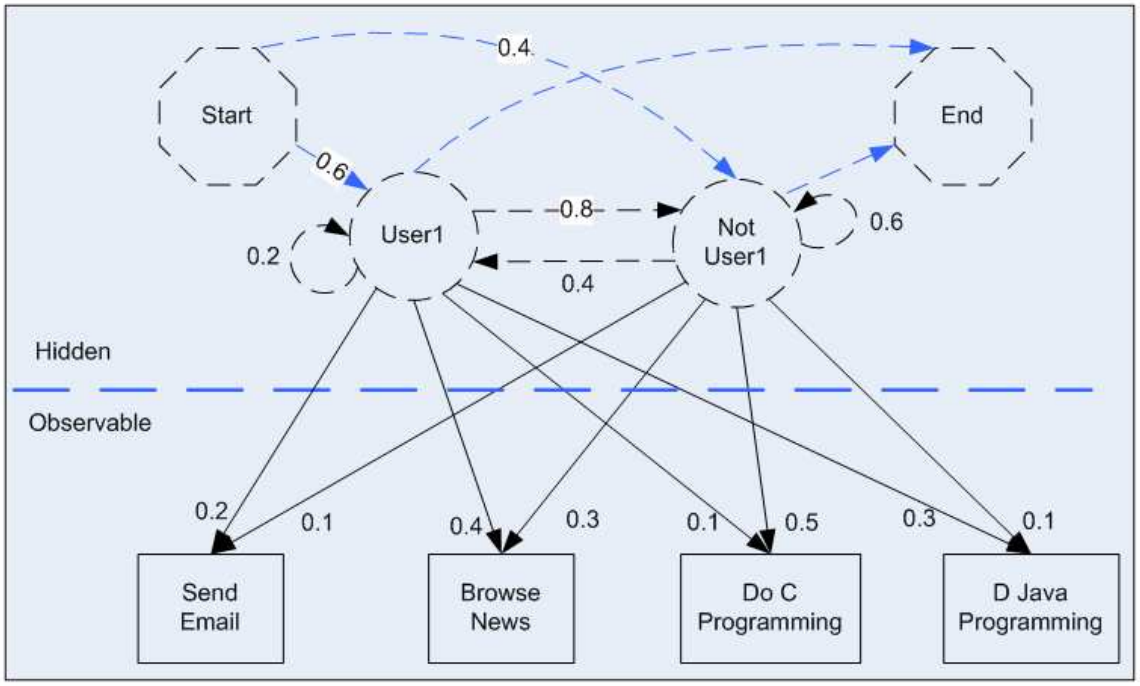

Figure 11: An HMM of a computer sharing pattern

Although we cannot directly see who is in front of the computer, we can observe the command sequences issued by the user. In this example, the commands are the observations. Suppose that we can characterize the user behavior patterns by analyzing the user history command sequences. For example, we have knowledge of what commands each user usually uses and the frequency of each command being used. As shown in Figure 11:, an arrow with a probability value is used to represent the relationship between a state and an observation. For example, the probability of User1 to issue a send email command is 20%. An N*M observation matrix is used to represented

the probabilities of all observations issued by states, where N is the number of states and M, the number of observation symbols. The observation matrix, denoted by *B*, in the Figure 11:can be summarized as:

$$B = \begin{bmatrix} 0.2 & 0.4 & 0.1 & 0.3 \\ 0.1 & 0.3 & 0.5 & 0.1 \end{bmatrix} \tag{4.4}$$

To establish an HMM, we need one more matrix π to indicate the initial state distribution. The initial state distribution in the Figure 11: is

$$\pi = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix} \tag{4.5}$$

Once we have the state transition probability matrix *A*, observation matrix *B*, and the initial state distribution matrix π, we are ready to define an HMM. Before we show the definition of an HMM, let us first look at the following notation [12].

Let

T = the length of the observation sequence

Q = $\{q_0, q_1, \ldots, q_{N-1}\}$ = the states of the Markov process

V = $\{0, 1, \ldots, M-1\}$ = set of possible observations

N = |Q| = the number of states in the model

M = |V| = the number of observation symbols

A = the state transition probabilities

B = the observation probability matrix

π = the initial state distribution

O = $(O_0, O_1, \ldots, O_{T-1})$ = observation sequence.

The observations are denoted by {0, 1, . . . ,M − 1}, and $O_i \in V$ for i = 0, 1, . . . , T−1.

Figure 12: illustrates a generic HMM [12]. The Markov process is determined by the initial state distribution matrix, π, and the state transition matrix, *A*. This process is hidden, and we can only observe the observation sequence. The observations are determined by the state transition matrix, *A*, and the observation probability matrix, *B*. An HMM can be defined by *A*, *B*, and π, and M, N implied by the matrices, i.e., $\lambda = (A, B, \pi)$ [12].

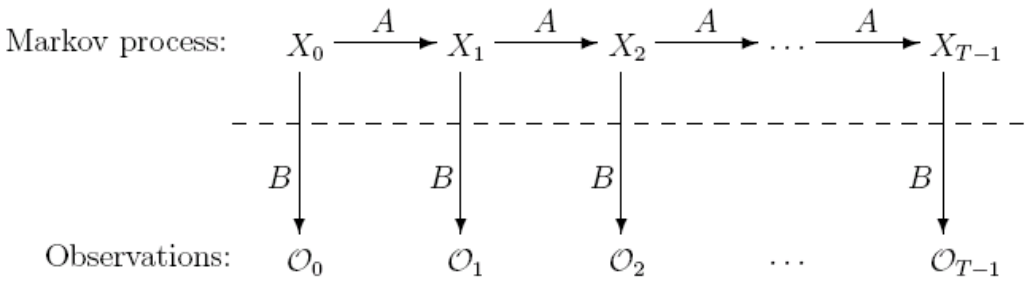

Figure 12: A generic HMM

An HMM can be used to solve three types of problems [12].

Problem 1: Determine the likelihood of an observed sequence O. In this problem, the input is an HMM $\lambda = (A, B, \pi)$ and O; and the output is P(O|λ).

Problem 2: Reveal the hidden state sequence of an HMM. Here, the input is the same as Problem1, i.e., an HMM $\lambda = (A, B, \pi)$ and O; but the desirable output is to find an optimal state sequence.

Problem 3: Train an HMM to best fit the observations. The input is a given observation sequence O and the values of M and N. The output is to find the model $\lambda = (A, B, \pi)$ maximizing the probability of O.

4.3. Implementation

In this project, we have constructed an HMM using the training data (Problem 3), and then used the created model to compute the likelihood of the test data (Problem 1). A high probability score indicates similar characteristics between the training data and the test data, and thus the test data will be recognized as normal data. A low probability score, on the other hand, indicates significant difference between the training data and the test data, and therefore the test data will be recognized as intrusion data. Once we get the probability scores, we can compute false positive rates and false negative rates by gradually varying the threshold value.

4.4. Experimental Results

4.4.1. The Detection Results of HMMs with 2 States, 4 States and 6 States

When we build an HMM from the training data, the number of states is the parameter we can change. Since we cannot directly know how many states an underlying model of the training command sequence has, we have trained HMMs with 2 states, 4 states and 6 states. Figure 13: shows that these three HMMs yield almost the same detection results in the lower false positive rate region. The model with 2 states performs slightly better than the other two. Naturally, in the following sections when we compare HMMs with other models, we use the results of the HMM with 2 states.

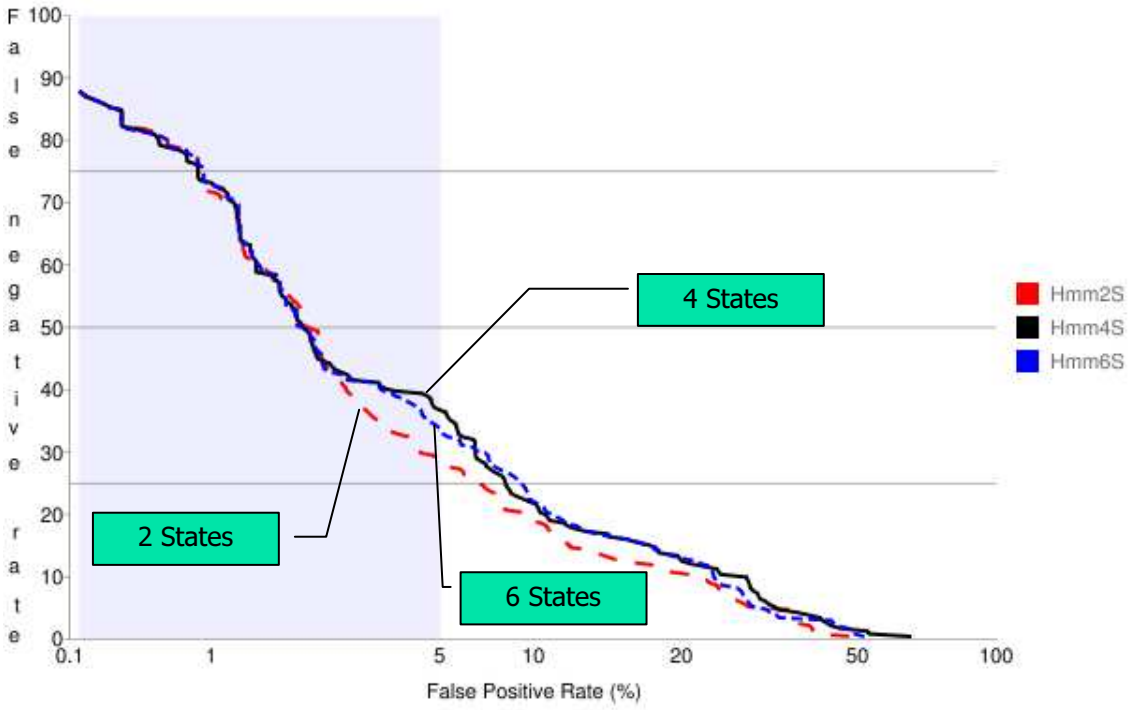

Figure 13: The detection results of HMMs with 2 states, 4 states, and 6 states.

4.4.2. HMM vs. N-Gram

Figure 14: shows the detection results of the HMM vs. those of the uniqueness weighted N-Grams. In the region of false positive rates between 0.1% and 1%, the HMM has a similar detection result as the uniqueness weighted 1-Gram. But the HMM performs better than the uniqueness weighted N-Grams in the region between 1% and 5%. As discussed in Section 3.2.3, the uniqueness weighted N-Grams yield best results provided by the N-Gram models. Thus, the HMM model outperforms the N-Gram models in the useful zone.

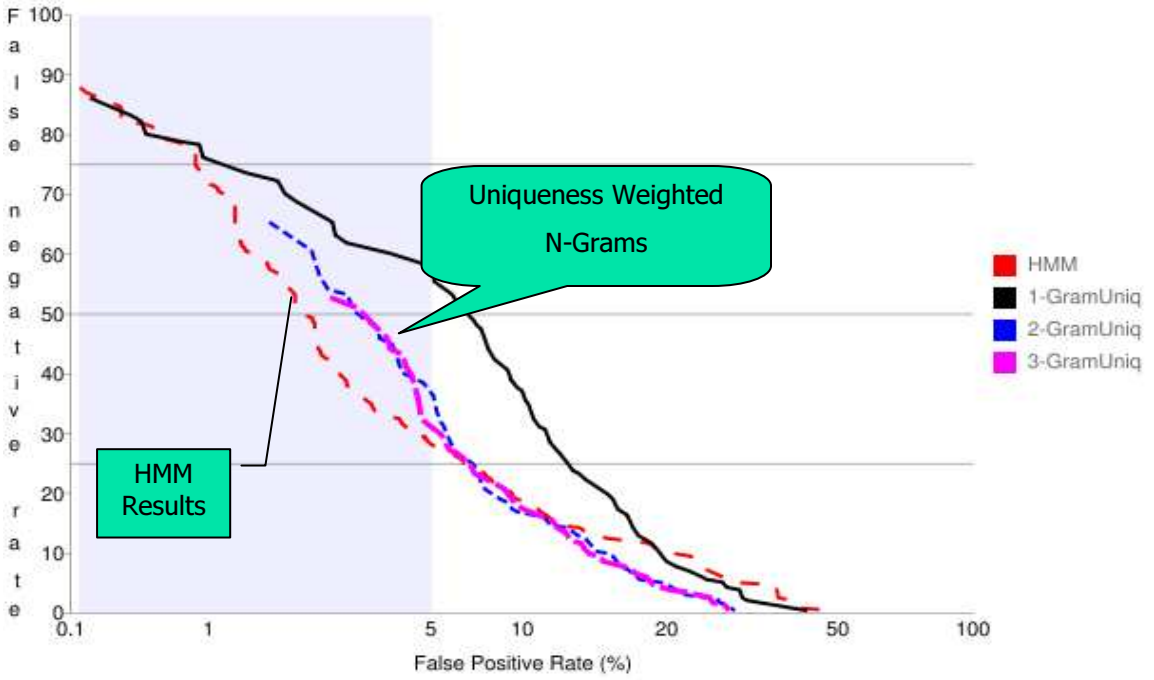

Figure 14: The detection results of the HMM vs. uniqueness weighted N-Grams

4.4.3. Conclusions

We have implemented an HMM for masquerade detection. We have conducted sensitivity analysis on the number of states for the HMM and conclude that the number of states has no significant effect on the detection results. We have also compared the detection results of the HMM with those of the uniqueness weighted N-Grams, the best results provided by the N-Gram models. We conclude that the HMM yields better detection results than the N-Gram models.

5. Profile Hidden Markov Model (PHMM)

5.1. Overview

A PHMM is a specific type of an HMM that adds an additional dimension of position in the original HMM. Specifically, a PHMM consists of a sequence of positions (or more precisely, states), and there is an HMM associated with each position.

A typical way of constructing a PHMM is to generate a multiple sequence alignment (MSA) from training data, and then to build the PHMM upon the MSA [15]. It takes several steps to obtain the MSA from training command sequences, and several more steps to build the PHMM on the MSA (see Figure 15: for the architecture of masquerade detection using a PHMM).

Here, we outline the steps of constructing a PHMM for masquerade detection (detailed procedure is described in the following sections):

1. Write a module to find the optimal pairwise alignments for two given command sequences:
    1.1. Generate a substitution matrix to provide the match/mismatch scores when aligning two symbols.
    1.2. Define a gap penalty function to measure the cost of aligning a symbol with a gap.
    1.3. Given the substation matrix and the gap penalty function for score calculation, use the dynamic programming algorithm to find the optimal local/global pairwise alignments with the highest score.

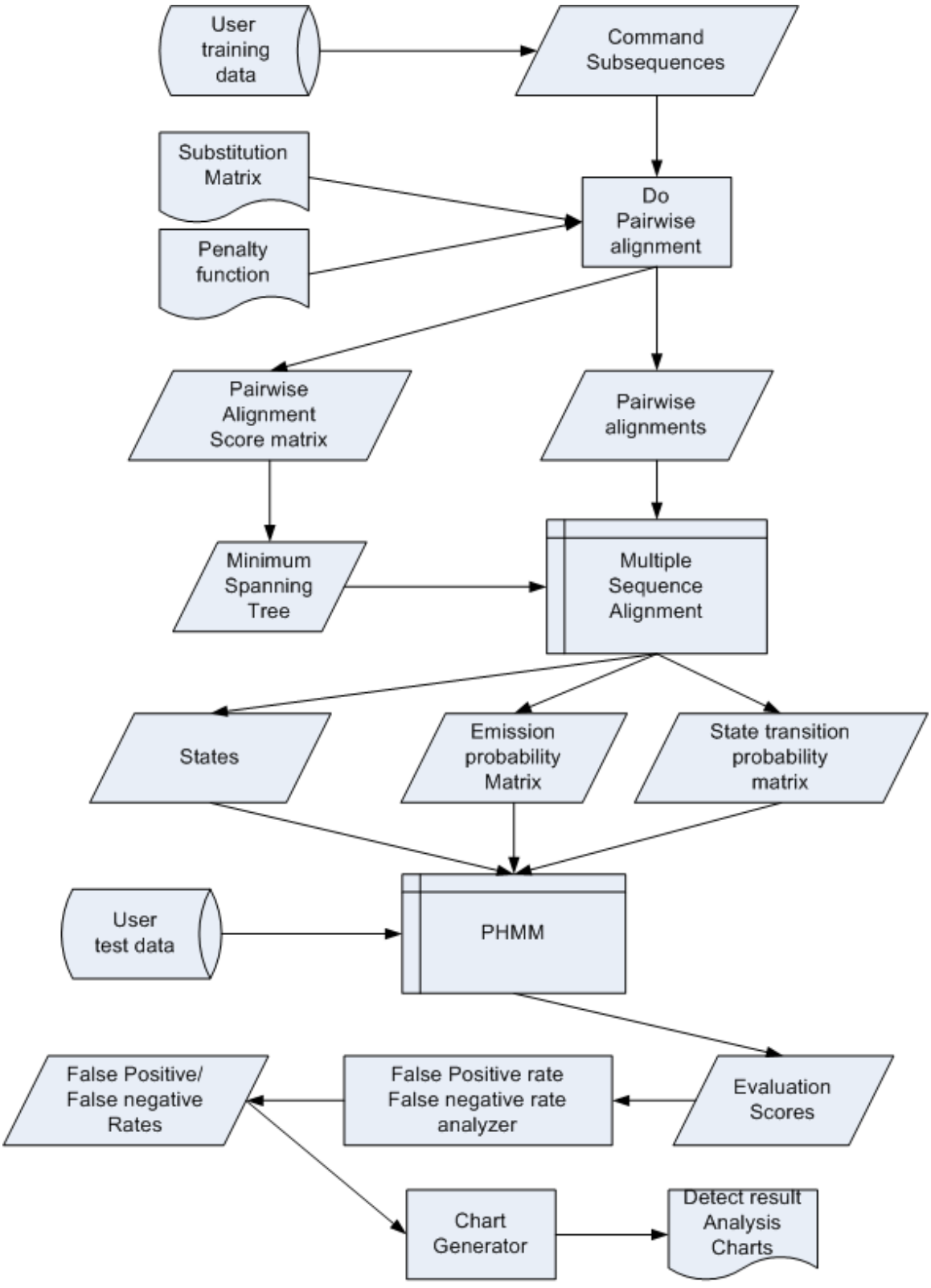

Figure 15: The architecture of masquerade detection using a PHMM

2. Generate the MSA from the training command sequences. Specifically, for each training command sequence, perform the following operations:
    2.1. Divide the command sequence into n subsequences.
    2.2. Find the pairwise alignments for all possible pairs amongst the command sequence and record their alignment scores in an n*n score matrix.
    2.3. Generate a minimum spanning tree from the score matrix. Designate one of the sequences with the highest pairwise alignment score as the root of the tree.
    2.4. Add subsequences to the MSA following the order that they are added to the minimum spanning tree.
3. Construct the PHMM using the obtained MSA.
    3.1. Determine the state for each position in the MSA.
    3.2. Calculate the emission probabilities for the states.
    3.3. Calculate the transition probabilities for the states.
4. Given a PHMM, calculate the probability for test data and analyze the detection results:
    4.1. Use the forward algorithm to score the test data.
    4.2. Compute false positive rates and false negative rates for different threshold values.
    4.3. Process the results and generate the charts

5.2. Implementation Details

5.2.1. Pairwise Alignment

When analyzing sequences, one of the most basic tasks is to find out whether two sequences are related [5]. Usually this task is divided into two steps:

1. Aligning the two sequences (this is often referred to as pairwise alignment),
2. Determining whether the two sequences are related based on the alignment results.

There are two types of pairwise alignments, local alignment and global alignment. Dynamic programming technique is the most commonly used method to find a pairwise alignment, since it guarantees to find the optimal match. Before using dynamic programming, we need to define a scoring model to compute the alignment score (or cost). Usually this is done by defining substitution matrices and gap penalty functions [5]. Substitution matrices are used to score the match and mismatch of two symbols. Gap penalty functions are used to measure the penalty for a symbol in one sequence to match to a gap in the other sequence.

5.2.1.1. Substitution Matrix

The basics idea of the dynamic programming algorithm is to perform command alignment by maximizing the alignment score. Therefore, we need a method to penalize a mismatch when the two aligning commands are not identical. A simple approach is to

treat all the mismatches as equally bad and then we can put a constant penalty for these mismatches. However, in reality, the effects of mismatches are often different and thus variable penalty should be considered. A typical way to implement variable penalty is to use an n*n substitution matrix S, where n is the number of the distinct commands used by the user. This matrix defines the scores of all possible pairs for a command to align to any other commands.

TABLE III: shows an example substitution matrix S based on Schonlau data set. A typical user in this data set uses 70 to 140 distinct commands. To simplify the example, we only consider 5 commands, say “send Email”, “Browse news”, “play Game”, “C programming”, and “Java programming”. These 5 commands are abbreviated as E, B G, C, and J, respectively.

TABLE III: AN EXAMPLE SUBSTITUTION MATRIX

	E	B	G	C	J
E	9	5	-4	2	2
B	4	8	-5	3	3
G	-4	4	9	-5	-4
C	2	2	-5	10	7
J	2	2	-5	7	10

In TABLE III:, the elements on the diagonal represent matches, and therefore have higher scores. The other elements represent mismatches, and thus have lower scores. Amongst the mismatches, we consider “C programming” and “Java programming” as closely related. Therefore, substituting “C programming” with “Java programming” receives small penalty with a high score of 7. In contrast, playing game is not closely related with programming and thus substituting “C programming” with “play Game” gets a high penalty with a low score of -5.

It should be noted that it is complicated to objectively define the relationship between any two UNIX commands. There are thousands of UNIX commands and to our knowledge there is no such study on the correlation amongst commands. Intuitively, each command has different significance for different user. Therefore, we have proposed to use the command significance to define the mismatch scores.

5.2.1.2. Gap Penalty

To generate a pairwise alignment, it is indispensable to have gaps unless the two given sequences are already optimally aligned. Hence, besides the match or mismatch, we should consider an additional case of aligning a command in one sequence to a gap in the other sequence. A natural question is how we should penalize the gaps. Intuitively, we should take in account the number of gaps and the length of each gap subsequence. There are two schemes for calculating the cost associated with an open gap [11]: linear score,

$$f(g) = -gd ,$$

and affine score,

$$f(g) = -d - (g-1)e ,$$

where g is the length of the gap, d is gap open penalty, and e is gap extension penalty. The linear score schema is a specific case of the affine score scheme with $d = e$. To penalize a new gap more than extending an existing one, we can give d a higher value

than *e*. Therefore, we have used the affine score for gap penalties in masquerade detection where some gaps in the sequences can be quite long.

5.2.1.3. Global Alignment and Local Alignment

There are two types of pairwise alignments to maximize alignment scores: global alignment and local alignment. Global alignment aligns every symbol while local alignment can align only the middle subsequence by discarding the beginning and ending subsequences with negative scores. Figure 16: lists two sequences and the corresponding global and local alignments.

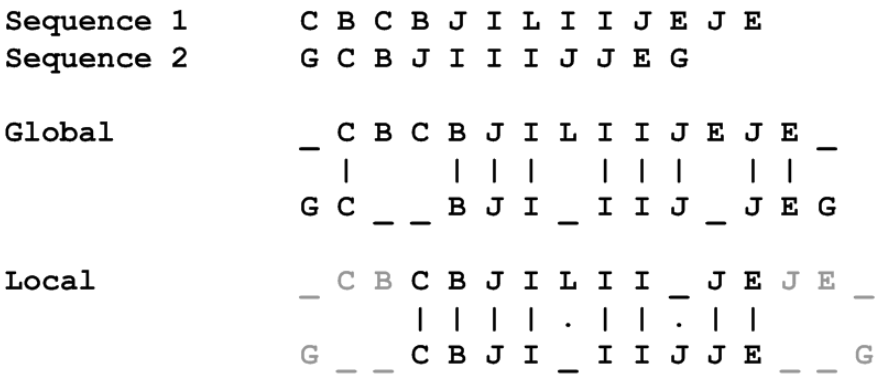

Figure 16: Global alignment and local alignment

Compared with local alignment, global alignment has an advantage of lossless information because every symbol is kept. However, global alignment may introduce too many gaps into the alignment, resulting in a less characterized alignment. Therefore, global alignment is suitable when two sequences are similar and have roughly equal lengths [17]. On the other hand, local alignment has an advantage of finding the most

common characterized subsequence when there is a significant difference between the overall characteristics of the two sequences. However, some significant information may be lost since local alignment may ignore the beginning and ending portions of the two sequences.

We have conducted some experiments on Schonlau training data and found that the command sequences have a low-degree similarity set. Therefore, we have used the local alignment to extract common features from the training data to generate the pairwise alignment.

5.2.2. Multiple Sequence Alignment (MSA)

As the name suggests, an MSA is an alignment of multiple sequences (See Figure 17: for an example of an MSA). A PHMM is constructed based on an MSA. This section describes the procedure of generating an MSA from Schonlau training data.

```
C _ _ _ _ B I B ...
_ C B C B B _ G ...
C B _ _ _ B _ G ...
I I _ G G B _ _ ...
I I G _ _ B G G ...
```

Figure 17: An example MSA

5.2.2.1. Preprocess Training Data to Get Multiple Sequences.

Schonlau training data provides a long list of 5000 commands for each user. To generate an MSA, we first need to obtain multiple sequences from this long list of raw data. We can divide the long list into multiple sequences by selecting some suitable dividing points. Intuitively, there is a tradeoff between the sequence length and the number of sequences. On one hand, too many sequences will generate numerous gaps in the alignments. On the other hand, if there are only a few relatively long sequences, then each state in the constructed PHMM has too few symbols to generate useful emission probabilities. We have generated 6 different multiple sequences based on the combination of the number of sequence and the length of the sequence as listed in TABLE IV:. These experimental results are provided and analyzed in Section 5.3.1

TABLE IV: EXPERIMENT CASES OF GENERATING MULTIPLE SEQUENCES

Case Number	Number of Sequence	Length of Sequence
1	4	1250
2	5	1000
3	8	625
4	10	500
5	20	250
6	50	100

5.2.2.2. Generate Pairwise Alignments

After obtaining multiple command sequences, we need to perform pairwise alignments for all possible pairs. We use the local alignment algorithm, substitution matrix, and penalty function defined in Section 5.2.1. In total, there are n*(n-1) alignments, where n is the number of command sequences. We also save the alignment

score of all the pairwise alignments in an n*n score matrix. The values on the diagonal of the score matrix are not used since we do not need to align a sequence to itself.

5.2.2.3. Generate MSA

We have implemented two different approaches to generate an MSA based on the pairwise alignments. The first approach is to add all the pairwise alignments into the MSA. In this approach, if there are n sequences, the MSA will contain n*(n-1) alignments.

Instead of adding all the pairwise alignments into the MSA at the beginning, the second approach is to gradually merge each sequence into the MSA. One solution for determining the order of adding the sequences to the MSA is to generate a minimum spanning tree from the score matrix. The sequence with the highest pairwise alignment score is designated as the root of the minimum spanning tree. Once we have the spanning tree, we can add sequences to the MSA in the order determined by the spanning tree. We have used Prim's algorithm to generate the minimum spanning tree.

5.2.3. Create PHMM

5.2.3.1. Determine MSA States

Figure 18: shows the architecture of a PHMM. A PHMM can be viewed as adding a position dimension into a standard HMM. At each position, there are three kinds of states: match, insert, and delete states. In Figure 18:, the match, insert, and delete states are represented by squares, diamonds, and circles, respectively. These three states correspond to the states in a standard HMM.

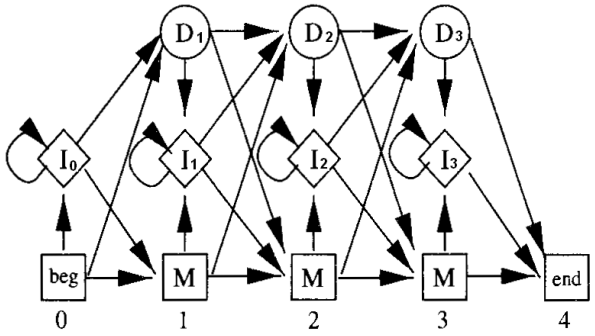

Figure 18: The architecture of a PHMM

Each symbol (e.g., commands in this project) in a PHMM belongs to either a match state or an insert state. A gap in a match state represents a deletion sate. To create a PHMM, we need to find out which columns (or positions) in the MSA form the match and insert states [15]. Columns with more symbols than gaps are considered as match states; otherwise, insert states [5]. Figure 19: shows an example of how to determine states for an MSA. While a match state consists of only one column (e.g. M1, M2…), an

insert state can contain multiple columns because the contiguous insert states are merged (e.g. I2).

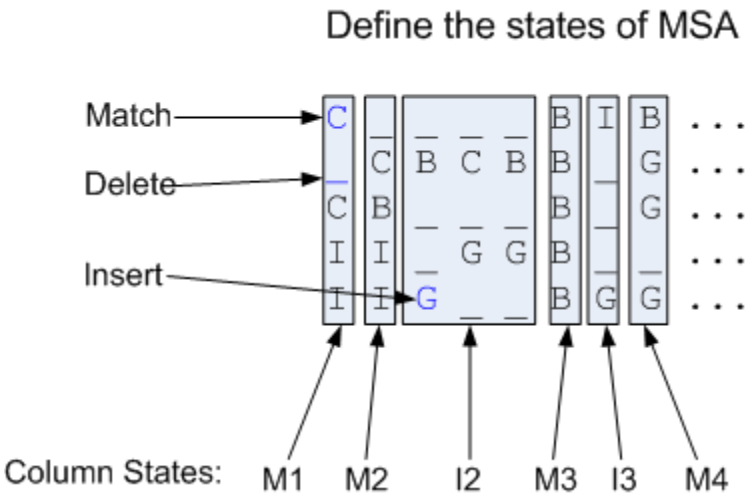

Figure 19: Determine MSA states

5.2.3.2. Calculate State Emission Probabilities

Each symbol in the MSA represents an emission. After the states are determined in the MSA, we can compute the state emission probabilities. For example, in Figure 19:, we can compute the probability of match state M_1 by counting the frequency for each symbol:

$e_{M1}(B) = 0/4$, $e_{M1}(C) = 2/4$, $e_{M1}(G) = 0/4$, $e_{M1}(I) = 2/4$.

To overcome the over-fitting problem, a common approach is to use "add-one rule" to eliminate the zero probabilities [5]. After applying the add-one rule, we have

$e_{M1}(B) = 1/(4+4) = 1/8$, $e_{M1}(C) = (2+1)/(4+4) = 3/8$,

$e_{M1}(G) = 1/(4+4) = 1/8$, $e_{M1}(I) = (2+1)/(4+4) = 3/8$.

Similarly, the emission probabilities for the insert state I_2 are calculated as:

$e_{I2}(B) = (2+1)/(6+4) = 3/10$, $e_{I2}(C) = (1+1)/(6+4) = 2/10$,

$e_{I2}(G) = (3+1)/(6+4) = 4/10$, $e_{I2}(I) = (0+1)/(6+4) = 1/10$.

Given an MSA, we compute the probabilities for all match and insert states and store the results in an emission matrix *E*. Matrix *E* corresponds to the Matrix *B* in the standard HMM, with the difference that Matrix *E* is position dependent. For an MSA with n match states, the matrix *E* consists the probabilities of $e_{M1}, e_{M2} \ldots e_{Mn}$, and the insert states of $e_{I0}, e_{I1} \ldots e_{In}$. For those insert states not presented on the MSA (such as, I_0 and I_1,), we assign each symbol with equal emission probability. For example,

$e_{I1}(B) = 1/4$, $e_{I1}(C) = 1/4$, $e_{I1}(G) = 1/4$, $e_{I1}(I) = 1/4$.

5.2.3.3. Calculate State Transition Probabilities

The state transition probability matrix *A* in a PHMM corresponds to the one in a standard HMM, with the difference that transition probabilities in PHMM are position-dependent. The matrix *A* contains all the transition probabilities from the begin state (denoted by M_0) to the end state (denoted by M_{n+1}). As shown in Figure 20: [5], *A* contains the information associated with all the arrows.

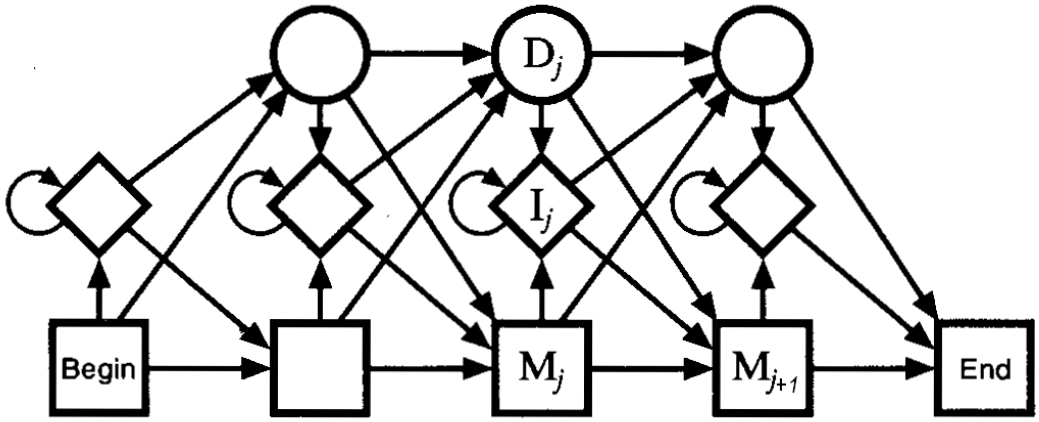

Figure 20: The state transition structure of a PHMM

The transition probability from state m to state n, denoted by a_{mn}, can be computed by dividing the total number of transitions from state m to any state by the number of transitions from state m to state n [5]. The add-one rule should also be applied by adding 1 for match, insert, and delete states. For example, the probabilities for the states transferring from match states M_1 are

$$a_{M1M2} = (3+1)/(4+3) = 4/7,\ a_{M1D2} = (1+1)/(4+3) = 2/7,\ a_{M1I2} = (0+1)/(4+3) = 1/7.$$

5.2.4. Calculate Test Data Probability and Detection Results

5.2.4.1. Forward Algorithm

Given a PHMM, we can use the forward algorithm to efficiently calculate the occurrence probability of an observation. Figure 21: provides the recurrence equation for the forward algorithm [5]:

$$
\begin{aligned}
F_j^{\mathrm{M}}(i) &= \log \frac{e_{\mathrm{M}_j}(x_i)}{q_{x_i}} + \log\left[a_{\mathrm{M}_{j-1}\mathrm{M}_j} \exp\left(F_{j-1}^{\mathrm{M}}(i-1)\right)\right. \\
&\qquad \left. + a_{\mathrm{I}_{j-1}\mathrm{M}_j} \exp\left(F_{j-1}^{\mathrm{I}}(i-1)\right) + a_{\mathrm{D}_{j-1}\mathrm{M}_j} \exp\left(F_{j-1}^{\mathrm{D}}(i-1)\right)\right]; \\
F_j^{\mathrm{I}}(i) &= \log \frac{e_{\mathrm{I}_j}(x_i)}{q_{x_i}} + \log\left[a_{\mathrm{M}_j\mathrm{I}_j} \exp\left(F_j^{\mathrm{M}}(i-1)\right)\right. \\
&\qquad \left. + a_{\mathrm{I}_j\mathrm{I}_j} \exp\left(F_j^{\mathrm{I}}(i-1)\right) + a_{\mathrm{D}_j\mathrm{I}_j} \exp\left(F_j^{\mathrm{D}}(i-1)\right)\right]; \\
F_j^{\mathrm{D}}(i) &= \log\left[a_{\mathrm{M}_{j-1}\mathrm{D}_j} \exp\left(F_{j-1}^{\mathrm{M}}(i)\right) + a_{\mathrm{I}_{j-1}\mathrm{D}_j} \exp\left(F_{j-1}^{\mathrm{I}}(i)\right)\right. \\
&\qquad \left. + a_{\mathrm{D}_{j-1}\mathrm{D}_j} \exp\left(F_{j-1}^{\mathrm{D}}(i)\right)\right].
\end{aligned}
$$

Figure 21: The recursion equation of the forward algorithm for a PHMM

Each notation of the above equation is described below:

i: the position in the observation sequence

j: the state position

$F_j^M(i)$: the probability of subsequence $x_0, x_1, \ldots, x_i$ on up to match state *j*

x_i: the *i*th observation symbol

$e_{Mj}(x_i)$: the emission probability of observing symbol x_i on match state M_j

q_{xi}: the probability of observing symbol x_i in a random model

The base case of the recursion $F_0^M(0)$ is initialized to 0. $F_j^I(i)$ and $F_j^D(i)$ are the probability of the subsequence of x_0 , x_1 ,…, x_i on up to insert and delete state *j*, respectively.

We use the above forward algorithm to score an observation sequence against a PHMM. Given a PHMM with *q* match states, the final score of an observation sequence with *p* symbols is defined as [15]:

$$\text{Score} = \log(a_{MqMq+1} \exp(F_q^M(p)) + a_{IqMq+1} \exp(F_q^I(p)) + a_{DqMq+1} \exp(F_q^D(p))) \quad (5.1)$$

We can use this equation to score the test data against the constructed PHMM and save the scores on the file.

5.3. Experimental Result

5.3.1. Detection Results of Different Subsequence in MSA

As discussed in Section 5.2.2.1, the training data need to be divided into multiple sequences. We have experimented with several different values of the number of sequences, ranging from 4 to 50. Figure 22: shows the experimental results. While the PHMM with 5 sequences in MSA yields the lowest false negative rates in the useful zone, the overall detection results for these values do not differ significantly.

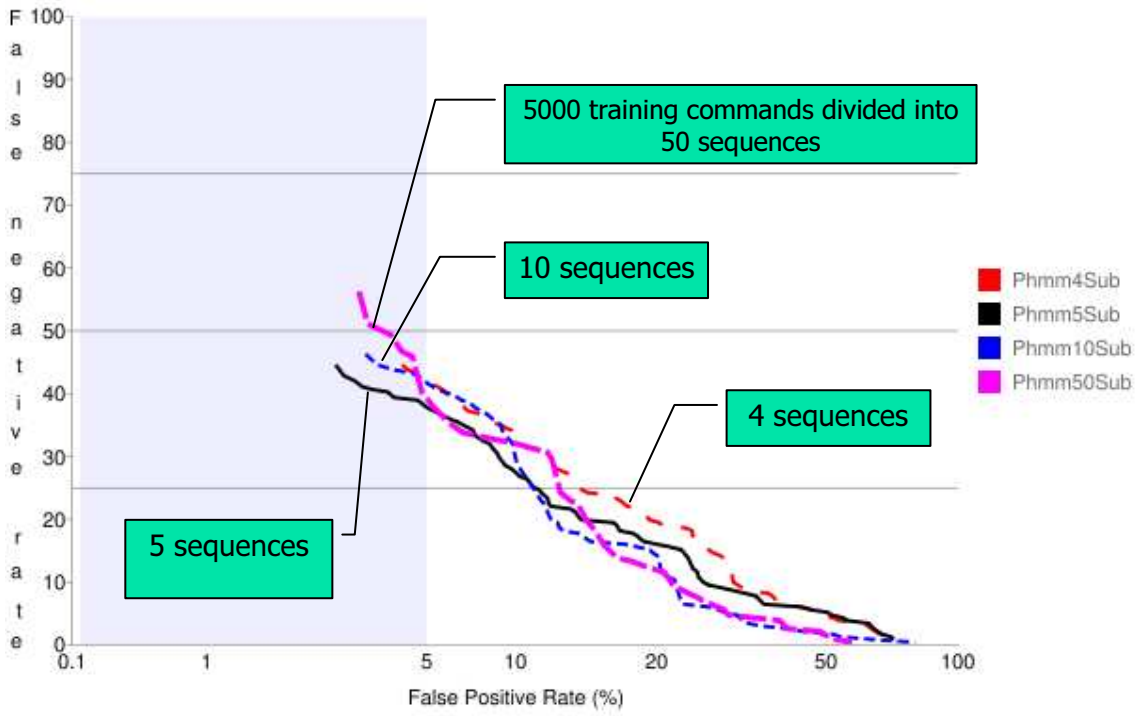

Figure 22: The detection results of PHMMs with different number of subsequences in MSA

5.3.2. PHMM vs. HMM vs. N-Gram Models

Figure 23: compares the detection results of PHMMs with those of the HMM and uniqueness weighted N-Gram Models. In the useful zone, the results of PHMM models are close to those of uniqueness weighted 3-Gram model, but not as good as those of the HMM.

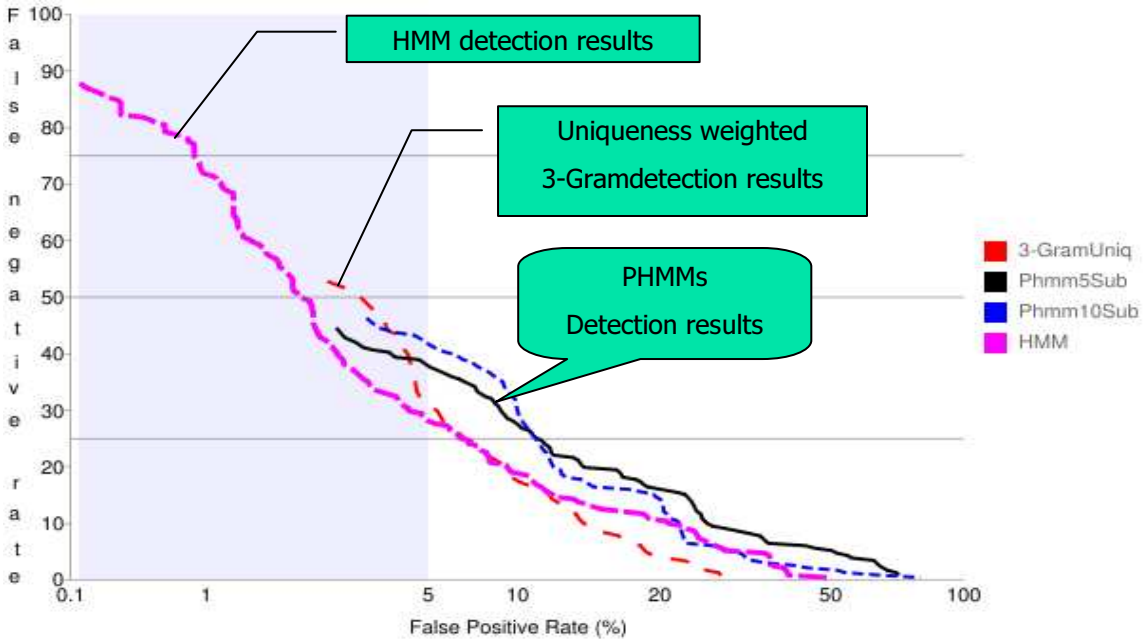

Figure 23: The detection results of PHMM models vs. the HMM model vs. the uniqueness weighted 3-Gram model

The reasons that the PHMM models do not yield better detection results than the HMM are multiple-fold. First, in Schonlau data set, there is no session beginning and ending information available on both the training data and the test data. A PHMM extensively relies on position information, and thus it is undesirable for the data to lack the session position information. Without position information, the PHMM would eventually degrade to HMM. Second, during the process of creating an MSA, some information is omitted by local alignment operations. Third, unlike protein sequences [5] or metamorphic viruses [15] where the evolved sequences are in fact from a common source, user command sequences do not have such a common source. Users might act on similarly as they previously do but the commands issued by them are not a modification of a common command sequence. We conjectured that the first reason had a major effect

on the poor performance of PHMMs. To confirm this conjecture, we have generated data sets with position information in the following section.

5.4. Generate Data Sets with Position Information

In order to stimulate user command sequences with session beginning and ending information, we have designed and implemented a user command sequence generator. For each user in Schonlau data set, we use a Markov chain to generate a training command sequence and "normal" command blocks in test data.

Firstly, to build such a Markov chain, we construct an initial state distribution matrix, denoted by π , and a state transition probability matrix, denoted by A. To calculate matrix π , we first count the number of distinct commands in the user training command sequence. Let n denote this number. Then we create an array sized n to store the frequencies of these commands. For matrix A, we create an n*n matrix to represent all possible transitions amongst the n distinct commands.

Secondly, we generate a "real-looking" user command sequence based on π and A. π is used to generate the first command, and A is used to generate the following commands. In our implementation, we sort the matrix π in the order of command frequencies. Only the first m most frequently used commands are selected as the candidates of the first command, where m is a configurable parameter of the command generator.

Finally, we randomly generate masquerade command sequence blocks. We have taken a block-based algorithm used by Schonlau, which randomly selects a block from Schonlau data set for the other users [8].

5.5. Detection Results of HMM vs. PHMM on Generated Data Sets

Recall that Schonlau training data consist of 5,000 commands and that the test data consist of 10,000 commands divided into 100 blocks. We have generated the same sized training data and test data. Then we use this whole generated set to construct a HMM and a PHMM. As shown in Figure 24:, the detection results of HMM (the black line) and PHMM (the light blue line) on our generated data set yields much better performance than those of the HMM on Schonlau data set (the red line). We can exclude the effect of masqueraded data, since we have generated the masqueraded data in the same way as Schonlau has. But we generate the training data using a Markov chain. Naturally, our training data is a better source for the HMM and the PHMM than Schonlau training data. In addition, since we have only selected the most frequently used commands as the candidates for the first command, our generated training data contains stronger characteristics than Schonlau does.

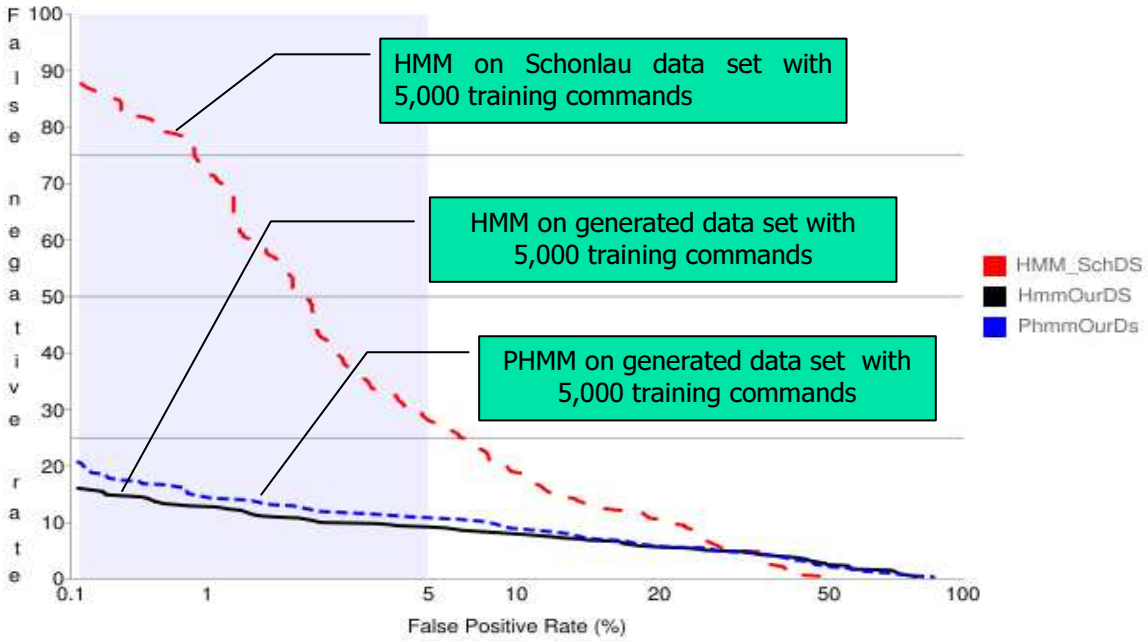

Figure 24: The detection results of the HMM and the PHMM on our generated data vs. Schonlau data set

However, the HMM still slightly outperforms the PHMM on our data set. We conjectured that our training data well represented user behavior patterns and thus the position information did not provide a significant advantage for a PHMM. To verify this conjecture, we have reduced our training data to boost the importance of the position information. Figure 25: shows the detection results of HMM and PHMM when the training data sets are reduced from 5,000 commands to 400 and 200 commands. Under such circumstances, the PHMMs significantly outperform the corresponding HMMs. In particular, the less the training data available, the better the PHMM performs than the HMM. As shown in Figure 25, the performance gain of the PHMM with 200 training commands over the corresponding HMM is significantly higher than the gain of the PHMM with 400 commands. This is because the position information plays a significant role in a PHMM when the training data does not sufficiently characterize user behavior.

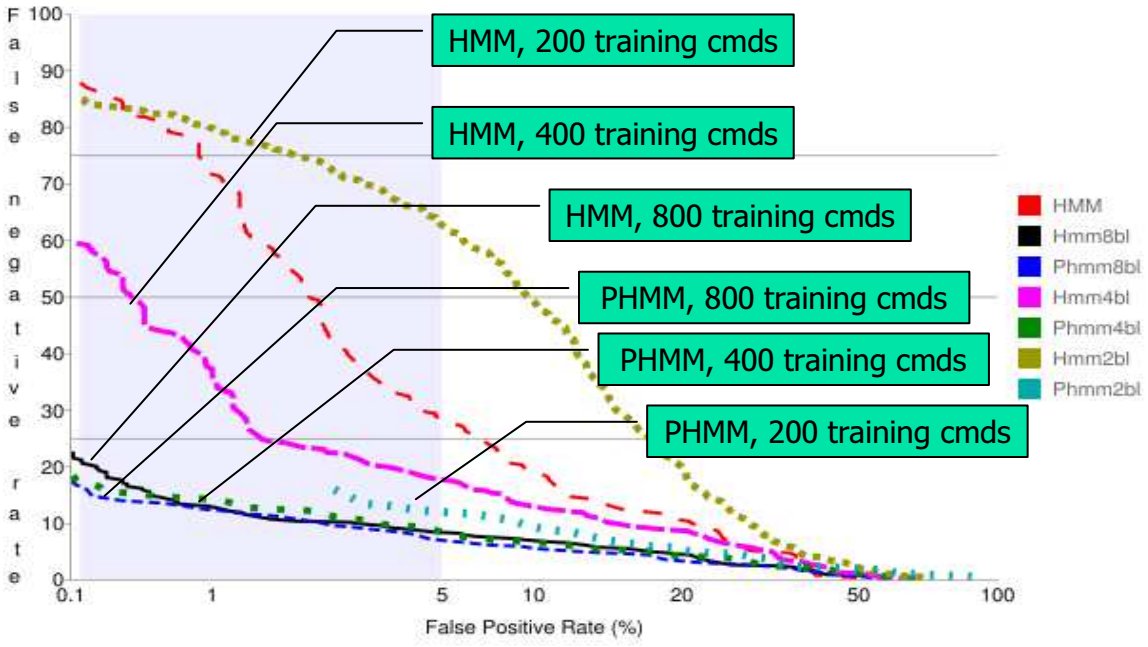

Figure 25: The detection results of the HMMs and the PHMMs on our generated data set with reduced training data

5.5.1. Conclusions

We have implemented the PHMMs for masquerade detection. We have established the substitution matrix and the penalty function, and created pairwise alignments using dynamic programming algorithm. We have also constructed an MSA from the training data, and built PHMMs using the generated MSA. Then, we have used established PHMMs to score the test data.

We have compared the detection results of the PHMMs with those of the HMMs and the uniqueness weighted N-Gram models using Schonlau data set. We have found that the PHMMs does not yield better performance than the HMMs since Schonlau data set lack position information required for the PHMMs.

To overcome the limitation of Schonlau data set, we have designed and implemented a user command sequence generator using a Markov chain. The newly generated data characterize user behavior well and thus the position information does not boost the performance of the PHMMs. Therefore, we have reduced the training data size to magnify the importance of the position information. We conclude that when there is no sufficient training data, the PHMMs significantly outperform the corresponding HMMs since the position information complements the insufficient training data.

6. Conclusions and Future Work

In this project, we have studied several models for masquerade detection. We have implemented the N-Gram models using N-Gram frequency statistics. We have also added weights of global statistics, such as command usage percentage and uniqueness, into the N-Gram model. After comparing the detection results of un-weighted N-Gram model with the weighted models, we have concluded that adding the global statistics into the model yields a positive affect.

We have also implemented the HMMs for masquerade detection, and have conducted the sensitivity analysis on the number of states in the HMMs. The experimental results show that the impact of the number of states is minor.

Finally, we have designed and implemented the PHMMs, a novel approach for masquerade detection. We have compared the detection results of the PHMMs with those of the HMMs and the uniqueness weighted N-Gram models. The experimental results show that the PHMMs do not perform as well as the HMMs on Schonlau data set. We have analyzed the reasons and conjectured that it was primarily caused by the lack of the session starting and ending information required by the PHMMs.

To overcome the limitation of Schonlau data set, we have generated a data set with the session starting and ending information. We have found that since our generated data well represents user behavior, adding session starting information does not provide a performance boost for the PHMMs. However, when we reduce the training data size, the additional position information is significantly helpful and thus the PHMMs yields much better detection results than the corresponding HMMs.

At present we have not studied the updated algorithms on Schonlau data set. In other words, once the HMMs or PHMMs are constructed using the training data, these models are not updated per users' new behaviors. Other studies on the same data set have yielded better detection results by dynamically updating user profiles [3]. Therefore, future research can be conducted to study how much performance gain can be obtained by exploring the updated algorithms.

REFERENCES

1. M. Whitmanl, H. Mattord. Principles of Information Security. Canada: Thomson, 2009. Pages 290 & 301

2. Mark Stamp, Information security: principles and practice, Wiley 2005

3. M Schonlau, W DuMouchel, Computer Intrusion: Detecting Masquerades.

4. TF-IDF, http://en.wikipedia.org/wiki/Tf–idf

5. Durbin, B. Eddy, S. Krogh, A., Mitchison, G., Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids. Cambridge University Press, Cambridge (1998)

6. Thomas Jacobsen, A Fast method for the Cryptanalysis of Substitution Ciphers, 1995

7. Edwin Olson, Robust Dictionary Attack of Short Simple Substitution Ciphers.

8. M Schonlau, Masquerading User Data, http://www.schonlau.net/intrusion.html

9. Greenberg Data Set, http://pages.cpsc.ucalgary.ca/~saul/wiki/pmwiki.php/Resources/DataSets

10. Receiver Operating Characteristics (ROC) curve, http://en.wikipedia.org/wiki/Receiver_operating_characteristic

11. Hidden Markov Model: http://en.wikipedia.org/wiki/Hidden_Markov_model

12. Mark Stamp, A Revealing Introduction to Hidden Markov Models, http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf

13. D. Geng, T. Odaka, An N-Gram and STF-IDF model for masquerade detection in a UNIX environment, Springer 2010

14. M Latendresse, Masquerade Detection via Customized Grammars, Springer, Berlin 2005

15. S. Attaluri, S. McGhee, M. Stamp, Profile Hidden Markov Models and metamorphic virus detection, www.cs.sjsu.edu/faculty/stamp/students/Srilatha_cs298Report.pd

16. Intrusion Detection Systems - INTRODUCTION, DETECTION METHODOLOGIES: http://encyclopedia.jrank.org/articles/pages/6646/Intrusion-Detection-Systems.html

17. Sequence Alignment, http://en.wikipedia.org/wiki/Sequence_alignment

18. The GNU Accounting Utilities, http://www.gnu.org/software/acct/

19. S. Peleg and A. Rosenfeld, "Breaking substitution ciphers using a relaxation algorithm," Commun. ACM, vol. 22, no. 11, pp. 598–605, 1979.

20. R. Spillman, M. Janssen, B. Nelson, and M. Kepner, "Use of a genetic algorithm in the cryptanalysis of simple substitution ciphers," Cryptologia, vol. XVII, no. 1, pp. 31–44, 1993