# MASTER'S THESIS

On-line Adaptive IDS Scheme for Detecting Unknown Network Attacks using HMM Models

*by Irena Bojanic*
*Advisor: Dr. John S. Baras*

**SEIL MS 2005-1**
**(ISR MS 2005-1)**

INSTITUTE FOR SYSTEMS RESEARCH

*ISR develops, applies and teaches advanced methodologies of design and analysis to solve complex, hierarchical, heterogeneous and dynamic problems of engineering technology and systems for industry and government.*

*ISR is a permanent institute of the University of Maryland, within the Glenn L. Martin Institute of Technology/A. James Clark School of Engineering. It is a National Science Foundation Engineering Research Center.*

***Web site http://www.isr.umd.edu***

# ABSTRACT

Title of thesis: ON-LINE ADAPTIVE IDS SCHEME FOR DETECTING UNKNOWN NETWORK ATTACKS USING HMM MODELS

Irena Bojanic, Master of Science, 2005

Thesis directed by: Professor John S. Baras
Department of Electrical and Computer Engineering

An important problem in designing IDS schemes is an optimal trade-off between good detection and false alarm rate.

Specifically, in order to detect unknown network attacks, existing IDS schemes use anomaly detection which introduces a high false alarm rate. In this thesis we propose an IDS scheme based on overall behavior of the network. We capture the behavior with probabilistic models (HMM) and use only limited logic information about attacks. Once we set the detection rate to be high, we filter out false positives through stages. The key idea is to use probabilistic models so that even an unknown attack can be detected, as well as a variation of a previously known attack. The scheme is adaptive and real-time.

Simulation study showed that we can have a perfect detection of both known and unknown attacks while maintaining a very low false alarm rate.

# ON-LINE ADAPTIVE IDS SCHEME FOR DETECTING UNKNOWN NETWORK ATTACKS USING HMM MODELS

by

Irena Bojanic

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2005

Advisory Committee:

Professor John S. Baras, Chair/Advisor
Professor Carlos A. Berenstein
Professor Virgil D. Gligor

© Copyright by

Irena Bojanic

2005

To my parents

# ACKNOWLEDGMENTS

I would like to thank my advisor, Professor John Baras for giving me a great opportunity to work on challenging and very interesting projects, as well as for his guidance and advice. I thank Professor Carlos Berenstein and Professor Virgil Gligor for agreeing to serve on my thesis committee.

I owe thanks to my friends from Distributed Immune Systems group, for their advice and valuable comments.

Special thanks goes to my special person, Janko Segrt who has always been there for me, fully supporting me in all my efforts.

Also, I would like to acknowledge financial support of my research work and graduate studies through the ARO DAAD190110494 grant with the University of Maryland, College Park.

# TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

# Chapter 1

# Introduction

## 1.1 Intrusion Detection System

Intrusion detection systems (IDS) are used in order to provide security to computer systems. There are many different types of IDS as it will be described in detail in the next section. In our solution, we propose an on-line, adaptive intrusion detection systems scheme for detecting unknown network attacks using HMM models and logic. First, we will introduce some commonly used terminology and different types of IDS.

## 1.2 Intrusion Detection System terminology

Intrusion Detection Systems (IDS) are constantly developing and growing and so is the terminology associated with it. In this section we will try to introduce some of the most commonly used terminology [9].

**Alert**

An alert is a warning issued by the IDS to the system operator that an intrusion is taking place or being attempted. On detecting an intrusion, the IDS will alert the analyst using a variety of methods. If the console is local to the IDS the alert would normally appear on the monitor. The use of a warning sound can be used though on a busy IDS this solution would not be the most popular one. Alerts to a remote console can be sent using the vendor proprietary method (usually securely), email, SMS/Pager, or any combination of these methods.

### Anomaly

The majority of IDS will alert when certain events match the signature of a known attack. An anomaly based IDS will build a profile of the host or network activity over time. When an event occurs which is outside this profile the IDS will alarm, i.e. when someone does something they haven't done before. An example would be a user who suddenly gains administrator or root privileges.

### Attacks

Attacks can be considered attempts to penetrate a system or to circumvent a system's security in order to gain information, modify information or disrupt the intended functioning of the targeted network or system. The following is a list and explains the most common types of Internet attacks that an IDS is set up to detect.

### Attacks: DOS - Denial Of Service attack

Rather then penetrating a system security by hacking, a DOS attack will just take the system out, denying the service to its user. The means of achieving this are varied from buffer overflows to flooding the systems resources. Currently systems are slightly more aware of DOS, which results in more DDOS attacks.

### Attacks: DDOS - Distributed Denial of Service attack

A standard DOS attack, the type that uses large quantities of data from a single host to a remote host, can not deliver sufficient number of packets to achieve the desired result; therefore the attack will be launched from many dispersed hosts, hence the word 'distributed' in the name. Sheer weight of numbers takes out either the remote system or swamps it's connection.

### Attacks: Smurf

This is an older type of attack, but is still fequently attempted. A smurf occurs when a ping is sent to a smurf amplifiers broadcast address using the spoofed source address of the target; all the active hosts will then reply to the target, swamping the connection. Current top ten smurf amplifiers can be found at http://www.powertech.no/smurf/

### Attacks: Trojans

The term Trojan comes from the wooden horse used by the Greeks to attack Troy. The horse contained Greek soldiers who spilled out of the horse, after it was wheeled inside the city, and laid siege to the city and its inhabitants. In computer terms, it originally referred to software that appears to be legitimate, but that actually contains hidden malicious software. When this legitimate program is run, without any knowledge of the user, the malicious software is installed also. Since the majority of these malicious programs were remote control tools, the term Trojan soon evolved to refer to this type of tool, such as BackOrifice, SubSeven, NetBus, etc.

### Automated Response

Besides allerting to an attack, there are IDS that can automatically defend against them. This is done in different ways, like reconfiguring routers and firewalls to reject future traffic from the same address. However, problem with this method is that attackers can use a reconfigured device to their own advantage by spoofing the address of a friendly party and launching an attack; the IDS then configures the routers and firewalls to reject these addresses, making the attacker's attempt into a DOS attack. Another way to defend against attacks automatically is by injecting packets on the network to reset the connection. The problem with this method is there needs to exist an active interface, which would make itself susceptible to an attack. This would require that the interface is inside the firewall or using some other way around the problem.

### Exploits

For every vulnerability there is an exploit, a mechanism by which to exploit the vulnerability. An exploit can be considered the means of taking advantage of the structural weakness of the vulnerability. In order to attack a system, a hacker exploits vulnerabilities in the code.

### Exploits: Zero Day Exploit

A zero day exploit is an exploit that isn't known about in the wild, i.e. one that hasn't been caught yet. As soon as an exploit is discovered by the security world, it can be patched against and signatures can be written for IDS thereby making the exploit ineffective and the risk of being caught greater. Of course, zero day exploits are a very valuable commodity to hackers.

### False Negatives

A false negative occurs when an attack or an event is either not detected by the IDS or is considered not dangerous by the analyst.

### False Positives

A false positive is an event that is picked up by the IDS and declared an atack, but is actually a legal behavior of the system.

**Firewalls**

Firewalls are the network security door. They are not an IDS but their logs can provide valuable IDS information. A firewall works by rejecting unwanted connections based on rules of criteria, such as source address, ports, etc.

**Honeypot**

A honeypot is a system that can simulate one or many vulnerable hosts, providing an easy target for the hacker to attack. The honeypot should have no other role to fulfill, therefore all connection attempts are deemed suspicious. Another purpose of a honeypot is to take attackers attention from legitimate targets, and make them waste their time while the original entry point is secured.

## 1.3 IDS Categories

Since there are a lot of types of IDSs, there are also many different ways of how they work. The following is categorization of IDS based on type [9].

### 1.3.1 Application IDS

Application IDSs are aware of the intrusion signatures for specific applications, usually the more vulnerable applications such as webservers, databases etc. However, many of the host-based IDSs that ordinarily look at operating systems, have been becoming more application-aware. Many of the host-based IDSs that are not application-aware by default can be trained to become so. For example, a host-based IDS can provide information about everything that is going on from the event logs, including output from the event log reporting applications. Most of these events can be filtered by the operator because they have no security relevance, but there also exist events like viruses or failed accesses that can be given a higher priority.

An example of an application-specific IDS is Entercept Web Server Edition.

### 1.3.2 Consoles

In order to make an IDS suitable for the corporate environment, the dispersed IDS agents need to report to a central console. Nowadays many central consoles will also accept data from other sources, such as other vendors' IDSs, firewalls, routers etc. This information can be correlated to present a more complete attack picture. Some consoles will also add their own attack signatures to those supplied at agent level. Also, many of the consoles provide the facility of remotely administering the IDS.

Examples of this IDS category implementation are Network Security Monitor and Open Esecurity Platform.

### 1.3.3 File Integrity Checkers

When a system is compromised, attacker will often alter certain key files to provide continued access and prevent detection. By applying a message digest (cryptographic hash) to key files, the files can be checked periodically to see if they have been altered, thus providing a degree of assurance. Upon detecting such a change, the file integrity checker will trigger an alert. The same process can be employed by a system administrator after being successfully attacked, allowing the administrator to ascertain the extent to which the system has been compromised. Previously, file integrity checkers detected intrusions long after the event; however, more products have recently been emerging that check files as they are accessed, thereby introducing a near real time IDS element.

Some of the examples of this IDS category are Tripwire and Intact.

### 1.3.4 Honeypots

As mentioned in the IDS terminology section, a honeypot is a system that can simulate one or many vulnerable hosts, providing an easy target for the hacker to attack. The honeypot should have no other role to fill; therefore, all connection attempts should be deemed suspicious. Another purpose is delay: the attacker wastes time on the honeypot while the original entry point is secured, leaving the truly valuable assets alone.

Although one of the initial purposes of honeypots was to gather evidence for the prosecution of malicious hackers, there is much talk of entrapment when deploying honeypots; however, does the vulnerability of the honeypot give the hacker the right to attack it? In order to reach the honeypot an attacker would have had to circumvent at least one security device provided the honeypot is inside the attacked network.

Some examples of honeypots are Mantrap and Sting.

### 1.3.5 Host-based IDS

This kind of IDS monitors sys/event logs from multiple sources for suspicious activity. Host-based IDSs (also known as host IDS) are best placed to detect computer misuse from trusted insiders and those who have infiltrated the network evading traditional methods of detection. Basically, it is an event log viewer. A true host IDS will apply some signature analysis across multiple events/logs and/or time. Many will also incorporate heuristics into the product. Some will introduce an added benefit: because they operate at near real time, system faults are often detected quickly, which makes them popular with security personnel. The term host-based IDS has been applied to any kind of IDS sitting on a workstation/server. Vendors have tried this for various products from Network Node IDS to File Integrity Checkers.

Some of the examples of this IDS category are Kane Secure Enterprise and Dragon Squire.

### 1.3.6 Hybrid IDS

Modern switched networks have created a problem for intrusion detection operators. By default, switched networks don't allow network interface cards to fully operate in a promiscuous fashion (although some allow spanning ports or link mode Terminal Access Points (TAPs), whereby a certain TAP will see the traffic on all other TAPs.) However, some switches will not allow it at all, making the installation of a traditional network IDS difficult. Furthermore, high network speeds mean that many of the packets may be dropped by a NIDS. A solution has arisen in the form of Hybrid IDSs, which takes delegation of IDS to host one stage further, combining Network Node IDS and Host IDS in a single package. While this solution gives maximum coverage, consideration should be given to the amount of data and cost that may result. Many networks reserve hybrid IDS for critical servers.

Some of this IDS category examples are CentraxICE and RealSecure Server Sensor.

### 1.3.7 Network IDS (NIDS)

Network IDS monitors all network traffic passing on the segment where the agent is installed, reacting to suspicious anomaly or signature-based activity. Traditionally these were promiscuous packet sniffers with IDS filters, though these days they have to be far more intelligent, decoding protocols and maintaining state etc. They come in the form of appliance-based products that you just plug in to software that can be installed on computers. They analyze every packet for attack signatures, though under network load many will start to drop packets.

Many Network IDS have the facility to respond to attacks, the one that was covered under 'Automated Response' previously.

Some of this IDS category examples are SecureNetPro and Snort.

### 1.3.8 Network Node IDS (NNIDS)

Switched and/or high-speed networks have introduced a problem: some Network IDS are unreliable at high speeds, when loaded they can drop a high percentage of the network packets. Switched networks often prevent a network IDS from seeing passing packets. Network Node IDSs delegate the network IDS function down to individual hosts, alleviating the problems of both high speeds and switching.

While Network Node IDSs are closely related to personal firewalls, there are differences. For a personal firewall to be classed as an NNIDS, event analysis would have to be applied to the attempted connections. For instance, rather than ”attempted connection to port *****” as it is found on many personal firewalls, a NNIDS should identify a ”whatever” probe, applying a signature for the ”whatever” attack. A NNIDS would also pass events received at the host to a central console.

Some of NNIDS examples are BlackICE Agent and Tiny CMDS.

### 1.3.9 Personal Firewall

Personal firewalls sit on individual systems and prevent unwanted connections, incoming or outgoing. While not infallible, they are very effective in protecting hosts from attacks.

Some of the examples are ZoneAlarm and Sybergen.

### 1.3.10 Target-based IDS

This is one of those ambiguous IDS terms, which means different things to different people. One definition may refer to them being File Integrity Checkers, while an alternative is a network IDS that only looks for signatures of attacks to which the protected network may be vulnerable. The objective of the latter definition is to speed up the IDS by not looking for unnecessary attacks, which raises a question whether it is wise or not to exclude some attacks by default.

Chapter 2

# HMM problems

In our proposed solution of IDS, we use statistical models to characterize different behaviors of the system. The formal models representing normal and abnormal behavior of a system (in this case captured in the string of bsm data) are discrete Hidden Markov Models [1].

As described in [1], an HMM is an extension of Markov models to include the case where the observation is a probabilistic function of the state - the resulting model is a doubly embedded stochastic process with an underlying stochastic process that is not observable (it is hidden). An HMM is represented by a five-tuple $(S, V, A, B, \pi)$, the elements of which are described as follows:

- Finite set of possible states

  $$S = \{S_1, \cdots, S_N\}, \tag{2.1}$$

  where $S_i$ denotes the individual state, and $q_t$ denotes the state at time $t$. The number of states in the model is $N$.

- Finite set of possible observations

  $$V = \{v_1, \cdots, v_M\}, \tag{2.2}$$

  where the observation symbols correspond to the physical output of the system being modelled. The number of distinct observation symbols per state, i.e., the discrete alphabet size is $M$.

- Initial state distribution matrix $\pi = \{\pi_i\}$ , such that

$$\pi_i = Pr(q_1 = S_i), \qquad 1 \leq i \leq N. \tag{2.3}$$

- State transition probability distribution matrix $A = \{a_{ij}\}$ , such that

$$a_{ij} = Pr(q_{t+1} = S_j | q_t = S_i), \qquad 1 \leq i, j \leq N. \tag{2.4}$$

  In a special case where every state can be reached from any other state in a single hop, we will have $a_{ij} > 0$ for all $i, j$. For every other case, we would have $a_{ij} = 0$ for some pairs of $(i, j)$.

- Observation symbol probability distribution matrix for state $j$, $B = \{b_j\}$ , such that

$$b_j(k) = Pr(V_k \quad at \quad t | q_t = S_j), \qquad 1 \leq j \leq N, 1 \leq k \leq M. \tag{2.5}$$

For fixed $S$ and $V$, the parameters of an HMM are $\lambda = \{A, B, \pi\}$. Given appropriate values for $S, V, A, B$, and $\pi$, the HMM generates an observation sequence $O = O_1 O_2 \cdots O_T$ where $T$ is the number of observations in the sequence and each observation $O_t$ is a symbol from $V$.

## 2.1 Solving basic HMM problems

There are three basic problems that most applications reduce to. They are:

1. Given the model $\lambda = (A, B, \pi)$, how to compute $Pr(O|\lambda)$, the probability of the observation sequence $O = O_1 O_2 \cdots O_T$?

2. Given the model $\lambda = (A, B, \pi)$, how to choose a state sequence $Q = q_1 q_2 \cdots q_T$ so that $Pr(O, Q|\lambda)$, the joint probability of the given observation sequence $O = O_1 O_2 \cdots O_T$ and the state sequence given the model is maximized?

3. How to adjust the HMM model parameters $\lambda = (A, B, \pi)$ so that $Pr(O|\lambda)$ is maximized?

### 2.1.1 Problem 1.

The first problem is the evaluation problem, namely we want to calculate the probability of the observation sequence $O = O_1 O_2 \cdots O_T$, given the model $\lambda$, i.e. $Pr(O|\lambda)$. This problem corresponds to classification problem in our proposed IDS solution. In other words, we classify the observed sequence as a specific behavior by choosing the behavior model corresponding to the maximum probability of the observed sequence given the model.

We will briefly describe the computation process of this probability value; more details can be found in [1]. It is possible to find $Pr(O|Q, \lambda)$ for a fixed state sequence $Q = q_1 q_2 \cdots q_T$, then multiply it by $Pr(Q|\lambda)$ and then just sum up over all possible state sequences, Q's. Thus,

$$Pr(O|Q, \lambda) = b_{q_1}(O_1) \cdot b_{q_2}(O_2) \cdots b_{q_T}(O_T) \tag{2.6}$$

$$Pr(O|\lambda) = \pi_{q_1} a_{q_1 q_2} a_{q_2 q_3} \cdots a_{q_{T-1} q_T} \tag{2.7}$$

Hence, we have

$$Pr(O|\lambda) = \sum_{allQ} Pr(O|Q, \lambda) Pr(Q|\lambda) = \sum_{allQ} \pi_{q_1} b_{q_1}(O_1) a_{q_1 q_2} b_{q_2}(O_2) \cdots a_{q_{(T-1)} q_T} b_{q_T}(O_T) \tag{2.8}$$

However, this approach is computationally extensive, $2TN^T$ multiplications, i.e. even for small values, $N = 5$ and $T = 100$, we will have about $10^{72}$ multiplications [2], which would take a very long time even on a supercomputer to complete. Therefore, we use the forward-backward algorithm:

A new *forward* variable is introduced, $\alpha_t(i)$ and is defined as

$$\alpha_t(i) = Pr(O_1 O_2 \cdots O_t, q_t = S_i|\lambda), \tag{2.9}$$

the probability of the partial observation sequence up to time $t$ and the state $i$ at time $t$, given the model $\lambda$. Computation of $\alpha_t(i)$ is done by induction:

$$\alpha_{t+1}(j) = \Big[\sum_{i=1}^{N} \alpha_t(i) a_{ij}\Big] b_j(O_{t+1}), \qquad 1 \le t \le T-1, \quad 1 \le j, i \le N \tag{2.10}$$

where

$$\alpha_1(i) = \pi_i b_i(O_1). \tag{2.11}$$

The desired result is the sum of the terminal forward variables $\alpha_T(i)$:

$$Pr(O|\lambda) = \sum_{i=1}^{N} \alpha_T(i). \tag{2.12}$$

This approach involves calculations of the order of $N^2T$. It is useful to calculate a *backward* variable $\beta_t(i) = Pr(O_{t+1} O_{t+2} \cdots O_T | q_t = Si, \lambda)$,i.e. the probability of a partial observation sequence from $t + 1$ to the end, given state $S_i$ and the model $\lambda$, since it will be used in solving the Problem 3.

After initialization

$$\beta_T(i) = 1, \qquad 1 \le i \le N, \tag{2.13}$$

we iterate

$$\beta_t(i) = \sum_{j=1}^{N} a_{ij} b_j(O_{t+1}) \beta_{t+1}(j), \qquad t = T-1, T-2, \cdots, 1, \quad 1 \leq i \leq N. \tag{2.14}$$

### 2.1.2 Problem 2.

In this problem, objective is to find the "optimal" state sequence associated with the given observation sequence, i.e. to find $Q = q_1 q_2 \cdots q_T$ that will maximize $Pr(O, Q|\lambda)$. The algorithm that solves this problem is called the Viterbi Algorithm [3]. This is an inductive procedure that at each instant keeps the best possible state sequence (i.e. the one giving maximum probability) for each of the N states as the intermediate state for the desired observation sequence $O = O_1 O_2 \cdots O_T$. Finally, the path with the highest probability is selected. The algorithm proceeds as follows:

1. Initialization:

$$\delta_1(i) \quad = \pi_i b_i(O_1) \qquad 1 \leq i \leq N \tag{2.15}$$

$$\phi_1(i) \quad = 0. \tag{2.16}$$

where

$$\delta_t(i) = max_{q_1, q_2, \cdots, q_{t-1}} Pr(q_1 q_2 \cdots q_t = i, O_1 O_2 \cdots O_t | \lambda) \tag{2.17}$$

2. Recursion:

$$\delta_t(j) \quad = \quad max_{q \leq i \leq N} [\delta_{t-1}(j) a_{ij}] b_j(O_t), \quad 2 \leq t \leq T, \quad 1 \leq j \leq N \tag{2.18}$$

$$\phi_t(j) \quad = \quad argmax_{1 \leq i \leq N} [\delta_{t-1}(i) a_{ij}]. \tag{2.19}$$

3. Termination:

$$P^* = max_{1 \leq i \leq N}[\delta_T(i)] \tag{2.20}$$

$$q_T^* = argmax_{1 \leq i \leq N}[\delta_T(i)]. \tag{2.21}$$

4. Reconstruction:

$$q_t^* = \phi_{t+1}(q_{t+1}^*), \qquad t = T-1, T-2, \cdots, 1. \tag{2.22}$$

The resulting path, $q_1^*, q_2^*, \cdots, q_T^*$ solves the Problem 2. More detailed description of the Viterbi Algorithm can be found in [3].

### 2.1.3 Problem 3.

This problem, the most difficult one from the three, is to determine a method of adjusting the $\lambda$ parameters of an HMM to maximize the likelihood of the training set. In other words, we will use this method, called the Baum-Welch algorithm, to train our HMM models. This corresponds to the training process in our proposed IDS solution, when we want to create a model corresponding to the observed sequence.

An initial HMM model is assumed and is further improved by using the formulas below, in order to maximize $Pr(O|\lambda)$. An initial HMM can be constructed in any way, randomly or by some other estimation algorithm. The Baum-Welch algorithm maximizes $Pr(O|\lambda)$, also called the **likelihood function**, by adjusting the parameters of $\lambda$. This optimization criterion is called the **maximum likelihood criterion**.

A new variable, representing the probability of a trajectory being in state $S_i$ at time $t$ and making the transition to $S_j$ at $t+1$ given the observation sequence and model is given as

$$\xi_t(i,j) = Pr(q_t = S_i, q_{t+1} = S_j | O, \lambda) \tag{2.23}$$

We compute these probabilities using the forward backward variables:

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)}{Pr(O|\lambda)} \tag{2.24}$$

The probability of being in state $S_i$ at time $t$, given the observation sequence and the model is defined as:

$$\gamma_t(i) = Pr(q_t = S_i | O, \lambda) \tag{2.25}$$

which is obtained by summing over $j$:

$$\gamma_t(i) = \sum_{j}^{N} \xi_t(i,j). \tag{2.26}$$

Also, note that

$$\sum_{t=1}^{T-1} \gamma_t(i) = \text{expected number of transitions from} S_i, \tag{2.27}$$

and

$$\sum_{t=1}^{T-1} \xi_t(i,j) = \text{expected number of transitions from } S_i \text{ to } S_j. \tag{2.28}$$

Therefore, formulas for re-estimation of parameters of an HMM are given below:

$$\hat{\pi}_i = \gamma_t(i) = \text{ expected frequency (number of times) in state } S_i \text{ at time } t = 1 \tag{2.29}$$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \gamma_t(i)} = \frac{\text{expected number of transitions from state } S_i \text{ to state } S_j}{\text{expected number of transitions from state } S_i} \tag{2.30}$$

$$\hat{b}_j(k) = \frac{\sum_{t=1, O_t=k}^{T} \gamma_t(j)}{\sum_{t=1}^{T} \gamma_t(j)} = \frac{\text{expected number of times in state } j \text{ and observing symbol } v_k}{\text{expected number of times in state } j} \tag{2.31}$$

The procedure starts with the initial model $\lambda$. If the re-estimation model $\hat{\lambda} \neq \lambda$, and if $Pr(O|\hat{\lambda}) > Pr(O|\lambda)$, then we have found a better model from which the observation sequence $O = O_1 O_2 \cdots O_T$ is more likely to be produced. The procedure is repeated iteratively until $P(O|\lambda)$ is maximized, which solves Problem 3.

# Chapter 3

## Data set

Data set used to test the proposed IDS scheme was created at MIT Lincoln Laboratory and provided as part of the 1998 DARPA Intrusion Detection Evaluation [4].

The data is collected over a period of six weeks. The first two weeks should be considered as "alpha" training data collected as the simulation becomes fully instantiated and reaches a steady state [4]. Therefore this data should be used for initial training of IDS. The final four weeks of training data are more similar. They include more types of background traffic and attack variants then the first two weeks of data.

The first week of training data starts with a large amount of background traffic and average of two attacks per day. During the first and second week, services, attacks and additional traffic types are added until traffic patterns reach a final steady state by the end of the second week. The four remaining weeks will remain similar to each other with regard to attack types and background traffic. The six weeks of data includes more than 100 instances of 20 different attack types.

Data was collected continuously on a real network which simulated more than 1000 virtual hosts and 700 virtual users. There was BSM (Basic Security Module) `tcpdump` data. We used Solaris BSM audit records that contain a system call name, a set of arguments, process ID, user ID, etc, but the only information we used in simulations are system call name and payload associated with it.

## 3.1 Data transformation

In order to use the BSM data in the scheme, it was necessary to transfer it into a more suitable form for processing.

Since a BSM file contains the activity performed on machine `pascal` over one day, it includes both normal behavior of the system and attack traces. This big string is parsed into smaller strings containing 100 system calls each. The system calls are enumerated. Each file containing the parsed string is named as, for example, W3Monday.bsm.txt.split.10, which means that it is a 10th string of 100 system calls collected on Monday of the 3rd week of training.

Now, another pass is made through all the files and payload size associated with each system call is observed. This is done in order to differentiate between traces that contain only normal behavior of the system and traces that contain attack traces [1].

After completing this, another pass was made in order to collect traces of each attack, i.e. W5Thursday.eject.txt, which is a self explanatory file name. Then, the attack traces are parsed into strings of 100 system calls, containing name of the attack and sequence index, labeled as i.e. W5Thursday.eject.split.bad.342. Since in the window of time while an attack is going on there is also legal behavior of the

[1] Code developed by Trevor Vaughan, UMD

system, additional difficulty is present for detecting the attack - the probability of false negatives is high.

Now, since we need a sequence of numbers to use for HMM testing and training, we will map the system calls into numbers. What is specific to the buffer overflow attacks is payload size, and we want to somehow use that information. Therefore, we need to differentiate the system calls associated with huge payloads from the rest of the system calls. The algorithm passes through a file and assumes a number n for a system call (one-to-one mapping). If there is a payload greater than 300 associated with the specific system call, the number associated with that system call is 2*n. If the payload is less then 300, that system call is associated with the number 2*n-1. Therefore, if there is originally 60 different system calls, we will have 120 possible output numbers that will be used further.

## 3.2 Attack schedule

The following Table 3.1 describes the attacks we use in the scheme, that were included in training data of [4]. The first column indicates the week of the training and the second indicates the day. Third column shows starting time of the attack. All the attacks target the machine `pascal`. The last two columns specify the name and the variant of an attack. Keywords in the "Variant" column specify parameters and conditions for this instance of an attack. The keyword "clear" means that the attack was not made stealthy and that components of the attack should be visible in `tcpdump` and/or `bsm` data. The keyword "stealthy" means that attempts were made to hide components of the attack in the sniffing or audit data by encryption, by spreading the attack over multiple sessions, or by other techniques. Other keywords indicate characteristics of specific attacks or arguments for different attack programs.

Each of the attacks is explained in more detail in the following section.

## 3.3 The three attacks

The three attacks that will be used to evaluate our IDS scheme are `eject,` `ffbconfig` and `fdformat`. All of them belong to a class of User to Root exploits. In these exploits the attacker starts out with access to a normal user account on the system (maybe obtained by sniffing passwords, a dictionary attack, social engineering,..) and is able to exploit some vulnerability to gain root access to the system [5].

User to Root attacks cover several different types of attacks of which the most common one is the buffer overflow attack. Buffer overflow occurs when a program copies too much data into a static buffer without checking if the buffer can take that amount of data. All of the three attacks that will be described in detail here are types of a buffer overflow attacks.

Other User to Root attacks take advantage of poor temporary file management of some programs, some exploit race condition in the actions of a single program, or more programs running simultaneously.

| Week | Day | Time | Attack | Variant |
|---|---|---|---|---|
| 1 | Monday | 08:05:07 | format | clear |
| 1 | Monday | 08:07:13 | ffb | clear |
| 3 | Monday | 11:32:20 | ffb | clear |
| 4 | Friday | 09:22:12 | ffb | add .rhosts, stealthy |
| 4 | Friday | 10:55:44 | format | stealthy |
| 5 | Monday | 14:10:20 | ffb | ftp's over exploit files |
| 5 | Monday | 16:22:25 | ffb | chmod files |
| 5 | Monday | 17:47:29 | ffb | executes attack |
| 5 | Monday | 20:14:14 | format | clear |
| 5 | Tuesday | 14:43:08 | eject | clear |
| 5 | Tuesday | 16:39:11 | eject | clear |
| 5 | Wednesday | 22:42:37 | eject | stealthy |
| 5 | Friday | 08:50:38 | format | ftp over files |
| 5 | Friday | 12:34:29 | eject | run self contained exploit |
| 5 | Friday | 13:04:22 | format | chmod exploit files |
| 5 | Thursday | 09:06:46 | ffb | |
| 5 | Thursday | 09:32:03 | eject | |
| 5 | Thursday | 09:50:46 | eject | |
| 5 | Thursday | 10:00:14 | eject | |

Table 3.1: Attack schedule

### 3.3.1 Eject

**Description**

The `eject` program is used by some removable media devices that do not have an eject button or by removable media devices that are managed by Volume Management, in Solaris [5]. Since the bounds checking on arguments in the volume management library is insufficient, it is possible to overwrite the internal stack space of the `eject` program. In this way, the attacker can use this (buffer overflow) vulnerability to gain root access on attacked systems.

**Attack Signature**

An intrusion detection system can identify this attack by looking at the contents of the `telnet` or `rlogin` session the attacker is using (assuming that an attacker has already gained access to an account of the target machine and is running the exploit as a part of a remote session) and 'catching' one of the following events:

- assuming that an attacker transmits the C code to the target machine unencrypted, the IDS can scan for specific strings in the source code, i.e. when instead of the 'jump to' address, there is a string of unexpected characters (Ox%lxB[%d]..) or there is a line
  `execl('/bin/eject','eject',&buf(char*)0);`
- in case the attacker encrypts the source code, the 'jump to' address will have unexpected argument
- host based IDS can catch an eject attack by performing bottleneck verification

on the transition from normal user to root user and noticing that the user didn't make a legal user to root transition.

### 3.3.2 Ffbconfig

**Description**

The `ffbconfig` program configures the Creator Fast Frame Buffer (FFB) Graphics Accelerator, which is part of the FFB Configuration Software Package, SUNWffbcf. This software is used when the FFB Graphics Accelerator card is installed. As with `eject` attack, it is possible to overwrite the internal stack space of the `ffbconfig` program, since the bounds checking on arguments is insufficient.

**Attack Signature**

As is the case with the `eject` attack, the attacker who is exploiting the vulnerability has to transfer the code for the exploit onto the target machine, and then run the exploit. The IDS should look for the specific strings within the source code. A host based IDS can perform bottleneck verification or scan for the invocation of the "/usr/sbin/ffbconfig/" command with an oversized argument for the "-dev" parameter.

### 3.3.3 Fdformat

**Description**

The `fdformat` attack also exploits buffer overflow. The `fdformat` program formats diskettes and PCMCIA memory cards. This program also uses the same volume management library and is exposed to the same vulnerability as the `eject` program.

### Attack Signature

Similar to the two attacks described above, the attacker transfers the code for the exploit into the target machine and then runs the exploit. The IDS looks for specific strings within the source code, or invocation of the command "/usr/sbin/fdformat" with an oversized argument.

# Chapter 4

## Proposed IDS scheme: Introduction and Phase 1 - Initialization

In this chapter, we will describe in detail our proposed IDS scheme solution. If only misuse detection is used, there will be a lot of false negatives. On the other hand, if only anomaly detection is used, there will be a high rate of false positives. Therefore, we will try to introduce a hybrid solution, the one that satisfies the need for a good trade-off.

Since security is the goal, the main idea is to set the detection rate and then, through further analysis, minimize the false positive rate. We want good detection, low false positive rate, correct classification and detection of unknown attacks. Also, we want our IDS scheme to be on-line and adaptive.

The scheme consists of five phases: initialization, parallel testing and training, logic, verification and adaptive phase. In the first phase we construct HMM models for normal behavior and known attacks. By doing this, we actually partition 'probabilistic space' into three entities: normal behavior, known attacks and unknown attacks. In the second phase, we do the HMM multi-hypotheses testing of the incoming sequence against our models, while in parallel we also train current behavior model. In the third, logic phase, we take any suspicious sequences passed on by the previous phase and analyze it against logic models created using limited knowledge. In case that the sequence is not shown to be normal, we pass it on to

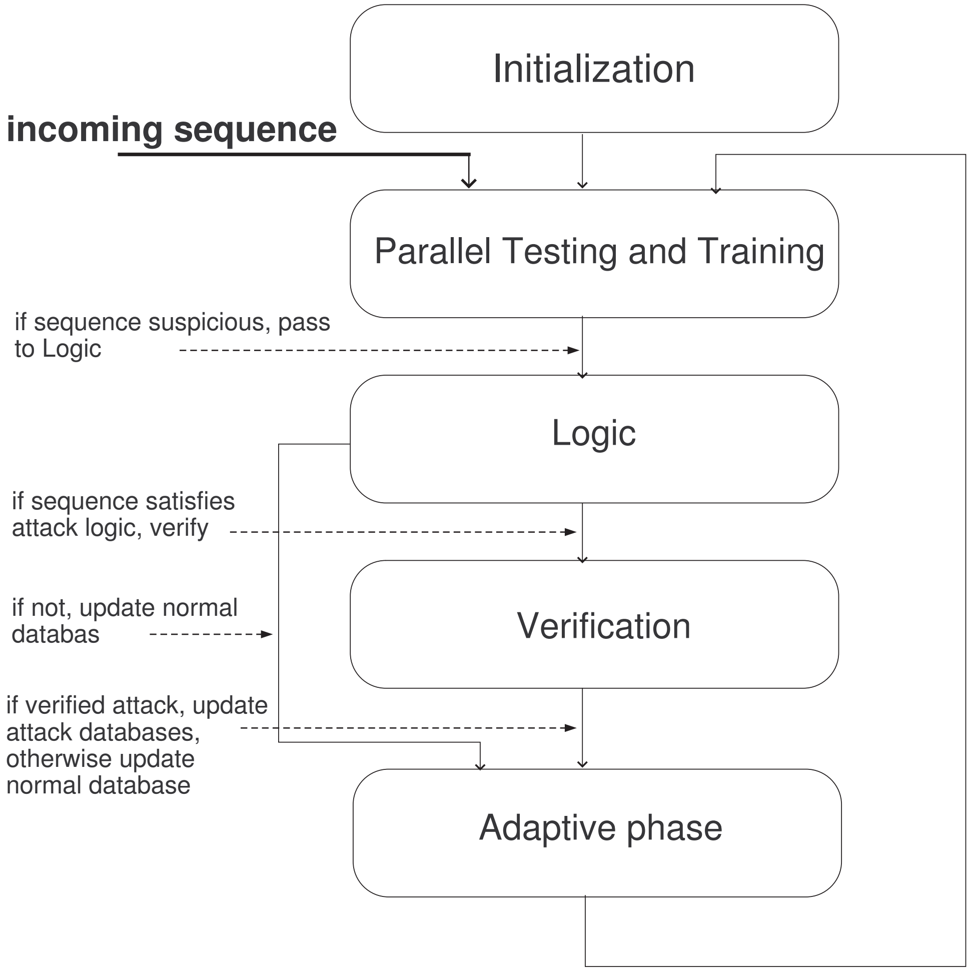
Initialization
incoming sequence
Parallel Testing and Training
if sequence suspicious, pass to Logic
Logic
if sequence satisfies attack logic, verify
if not, update normal databas
Verification
if verified attack, update attack databases, otherwise update normal database
Adaptive phase

Figure 4.1: Overall block scheme of the proposed IDS

the fourth phase where we verify if the sequence in fact is malicious and classify it correctly. The last phase updates our model databases. Through the whole scheme, there are several levels of alarms, (possibly) going off after each phase, hence alerting the administrator or triggering a possible automated response. The block scheme is shown in Fig. 4.1.

In this phase, the goal is to obtain models for all of the known attacks and a model for normal behavior, as described in Chapter 2. Note that, in order to evaluate our scheme, we will 'hide' at least one attack - it will be 'unknown', and, at the end, the scheme should be able to make a model for that one as well.

An HMM model is obtained by training an initial model with sequences obtained after transformation from BSM data. The model is trained sequentially with sequences of 100 system calls each, using the Matlab HMM Toolbox. After a relatively small number of iterations, log likelihoods for each model will stabilize.

We assume that states are unknown (hidden), and the only thing that we can observe is the sequence of numbers (produced by Data transformation algorithm). We start by assuming (random) transition and observation matrices. Then, we start the training process, by using the iterative algorithm that alters the transition and observation matrices so that at each step the adjusted matrices are more likely to generate the observed sequence (the training sequence). We have observed that parameters do not change significantly after a very small number of iterations (3), which is useful because of the run-time of the algorithm.

Assume that we know attacks `eject` and `ffbconfig`, but we have not seen `fdformat` yet. The ultimate goal of this scheme is to come up with an HMM model at the end for the `fdformat` attack as well.

The **Normal HMM** model is trained with attack-free sequences. However, since we want the scheme to also update this model in case there is normal behavior not previously seen, we will not use all the normal behavior sequences. For example, the **Normal HMM** model can be trained with normal sequences collected during first five weeks, and then, in the testing phase, we can use sequences from sixth week to check if the scheme works or gives false alarm.

The **Attack #1 HMM** model is trained with the sequences of that attack, for example, for the `eject` attack. The same procedure is done for **Attack #2 HMM** model and **Attack #3 HMM** model. We will use different combinations of known/unknown attacks, in order to evaluate the performance.

## 4.1 Normal Databases

In order to have maximal possible detection rate, with as small false alarm rate as possible, we need to make a good, extensive normal behavior database. However, in the data available, there is a huge number of normal sequences, and in order to have a good performance of our algorithm, we need to store it for future convenient and efficient use.

### 4.1.1 Normal HMM models

Since there are eight groups of collected sequences [4], each group representing a day in a specific week, we decided to make one model for each group. There are several reasons for that. First of all, if we had too many models, during simulation it would be impossible to have on-line detection, since memory access time would contribute a huge delay. Also, the issue would be storage. However, most of all, we would have to worry about overtraining. This means that our database would be 'too extensive', it would most probably cover much more virtual space, which we want to avoid, in order to be able to detect attacks, which are often embedded in the normal behavior sequences. For obtaining HMM models we used training process described in Chapter 2. In simulations, we used HMM MATLAB toolbox [8].

### Day 1

This group of sequences was collected on Monday of the first week of testing. It consists of 7330 sequences of length 100. Each sequence was used as an input to the HMM algorithm [8]. HMM has 5 states and 120 observations (each one for the mapping of a system call). Before training, the HMM has initial probability matrix [0.996,0.001,0.001,0.001,0.001), i.e. we start from the first state. Since every HMM model is characterized by 3 matrices: initial state probability matrix, transition matrix and observation matrix, we have to use them in order to represent accurately final representative model. Initial state probability matrix does not change much in the process and is not of a great importance for the future testing, so we will concentrate on the other two, transition probability (A) matrix and observation (B) matrix.

It is shown that there are little discrepancies between A matrices of these 7330 HMM models. This can be explained by the fact that during a course of one day, locality of time and locality of usage stereotypes the behavior. In the Fig. 4.2 we have shown 10 randomly chosen A matrices out of this group. It can clearly be seen that differences are minimal. Therefore, as a representative A matrix for the whole group we decided to use an average matrix of all 7330 A matrices, Fig. 4.3.

Analog logic is used in creating a representative B matrix for the group, instead

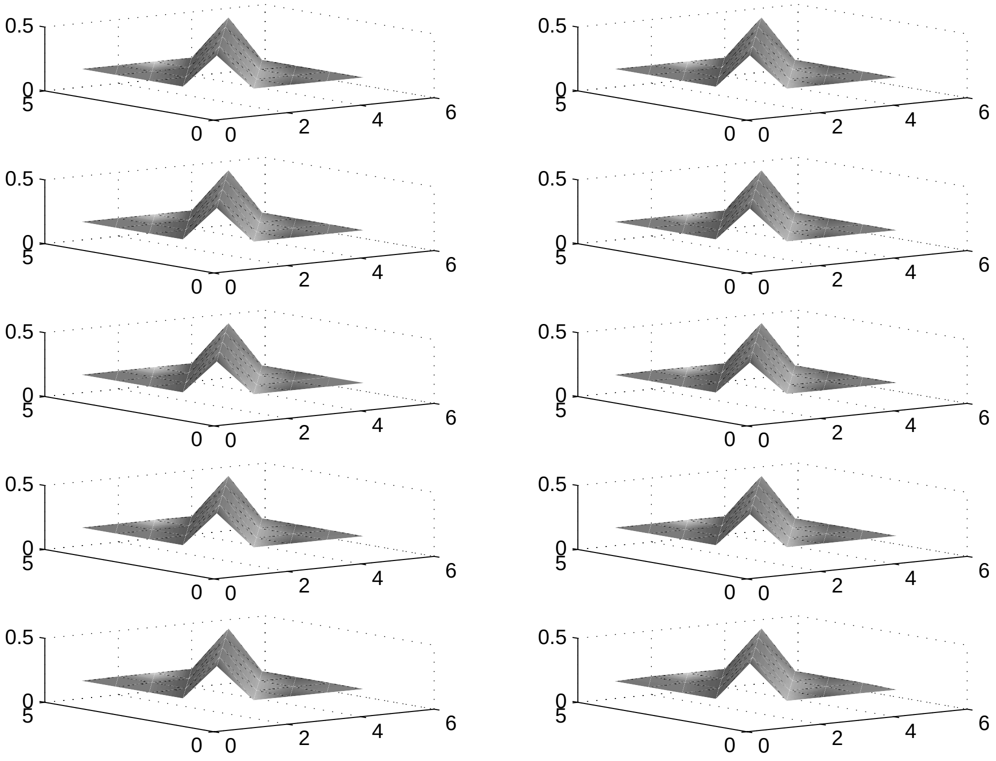

Figure 4.2: Transition matrices of 10 randomly chosen HMM's of Day 1

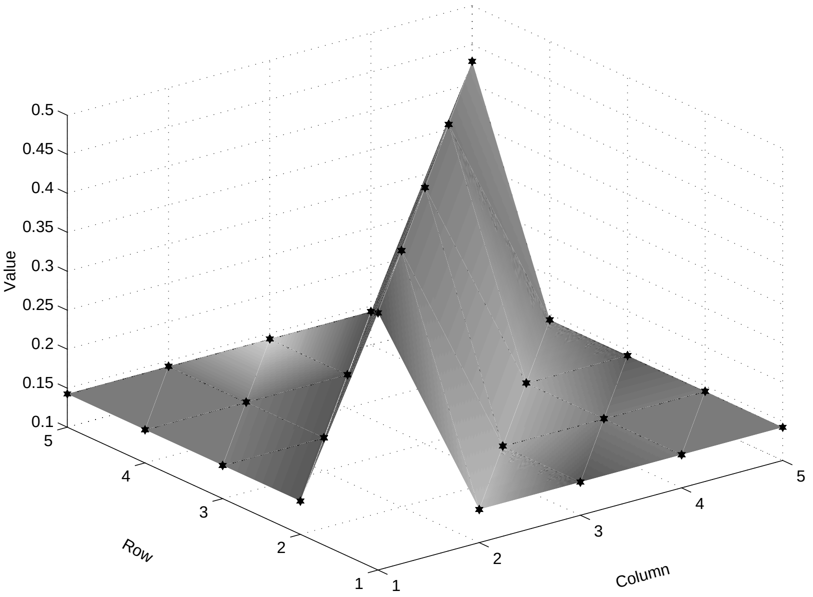

Figure 4.3: Transition matrix (A) for Day 1

that the B matrices are not as similar to each other as A matrices are, but we want to include every possible observable achieved by individual models, with weight proportional to its weight in individual matrices and to the number of individual matrices that contain it. Therefore, the averaging is fully justified and needed. In the Fig. 4.4 we are showing 10 randomly chosen B matrices from this group. Also, in Fig. 4.5 we've shown the average, representative matrix for the first normal HMM model that we will use in our Normal Database.

## Day 2

Using the same procedure as for the first group of normal behavior sequences, we have created a representative HMM model for the second group. This group consists of 19332 sequences of length 100, and is collected on Monday of the third

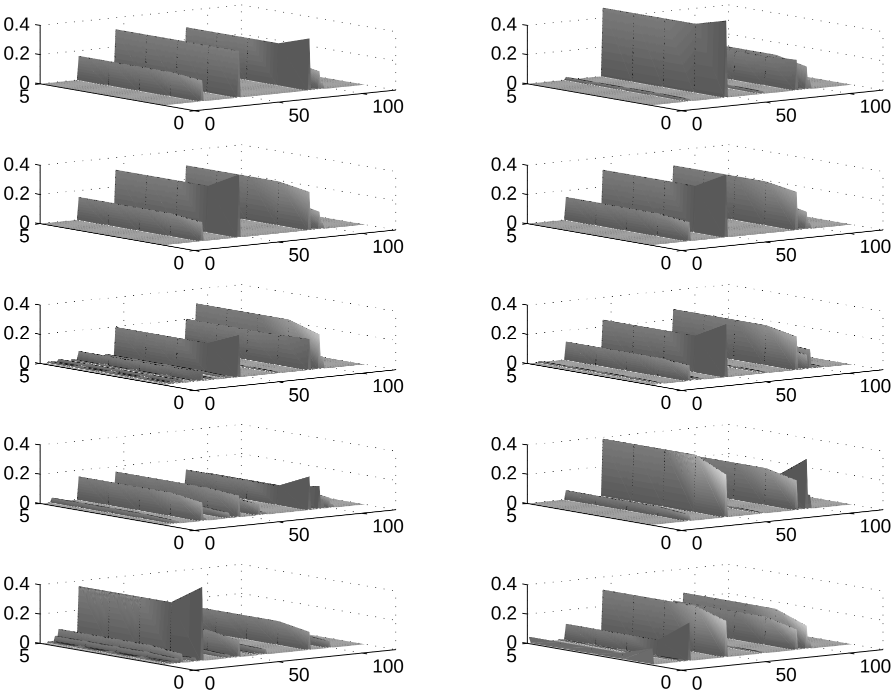

Figure 4.4: Observation matrices of 10 randomly chosen HMM's of Day 1

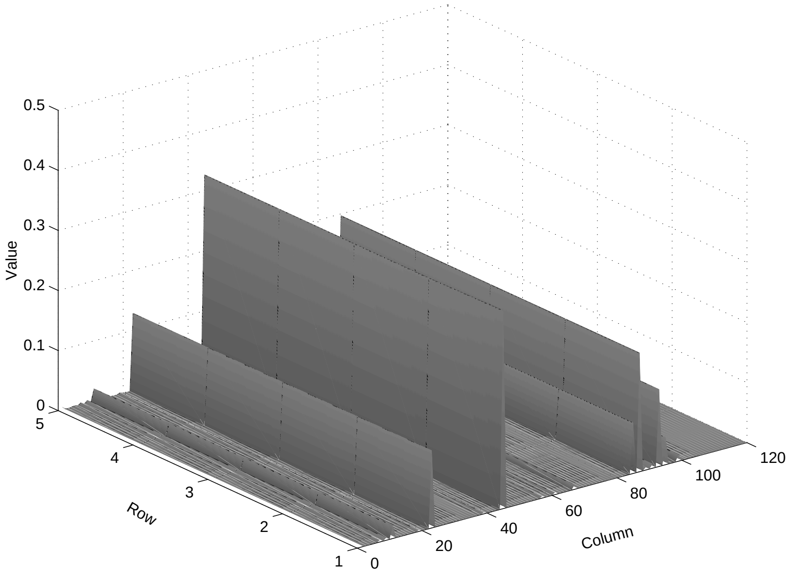

Figure 4.5: Observation matrix (B) for Day 1

week of collecting data [4].

The resulting model's A matrix is shown in Fig. 4.6.

Also, the representative matrix B for the second group is obtained using the same procedure as for the previous group and is shown in Fig. 4.7.

## Day 3

Third group consists of 19529 sequences of length 100, collected on Friday of the fourth week of collecting data.

The representative model's A matrix is shown in Fig. 4.8.

Also, the representative matrix B for this group is shown in Fig. 4.9.

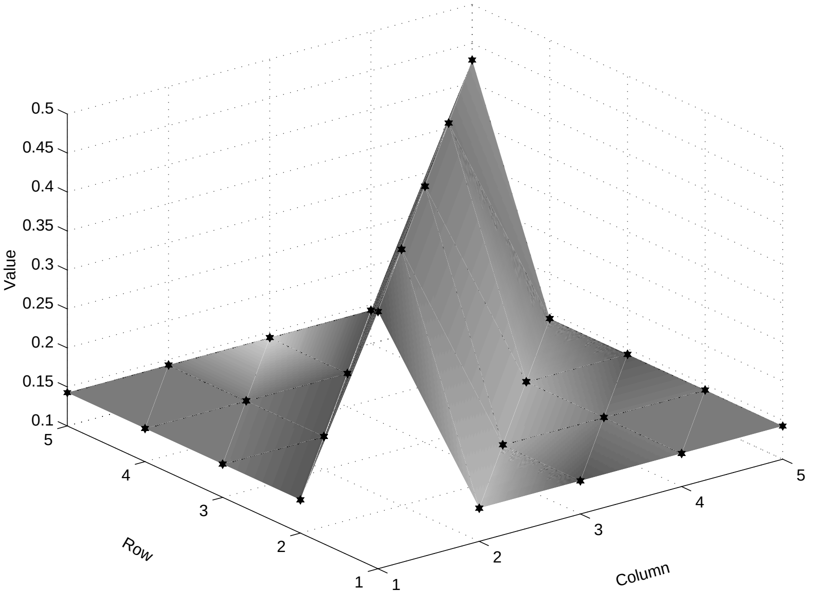

Figure 4.6: Transition matrix (A) for Day 2

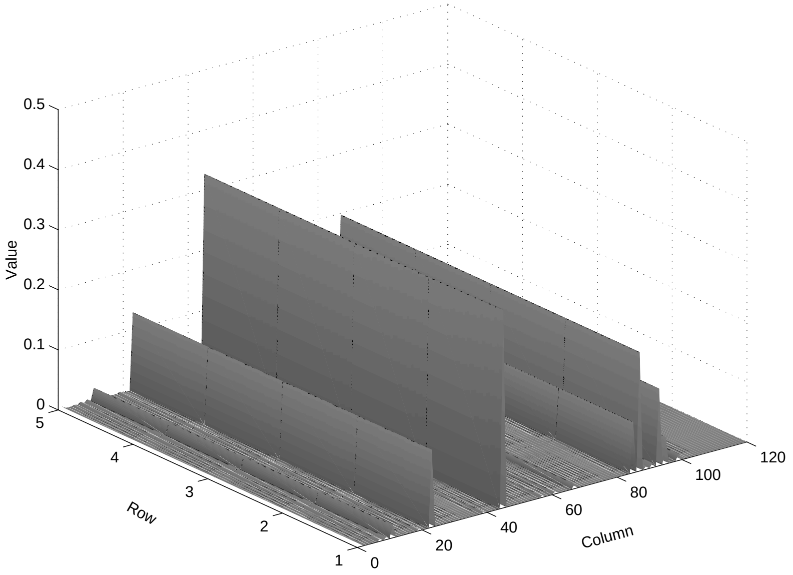

Figure 4.7: Observation matrix (B) for Day 2

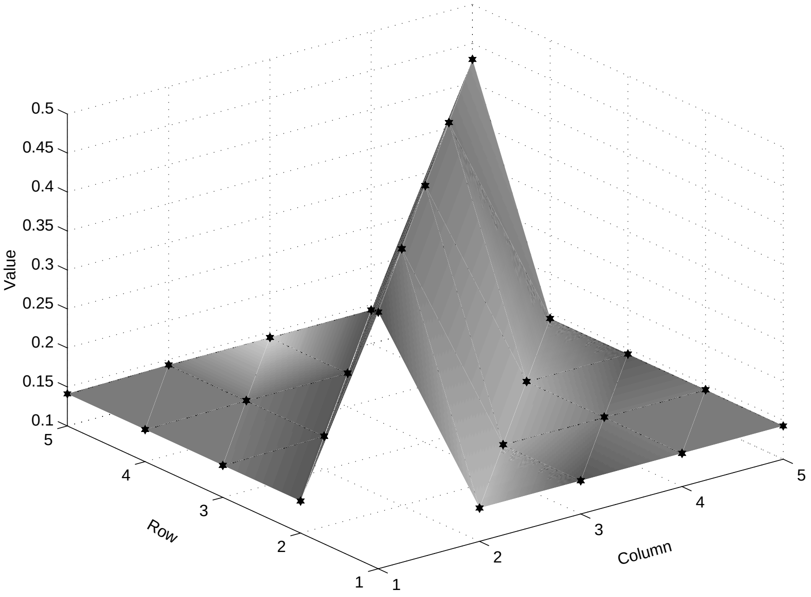

Figure 4.8: Transition matrix (A) for Day 3

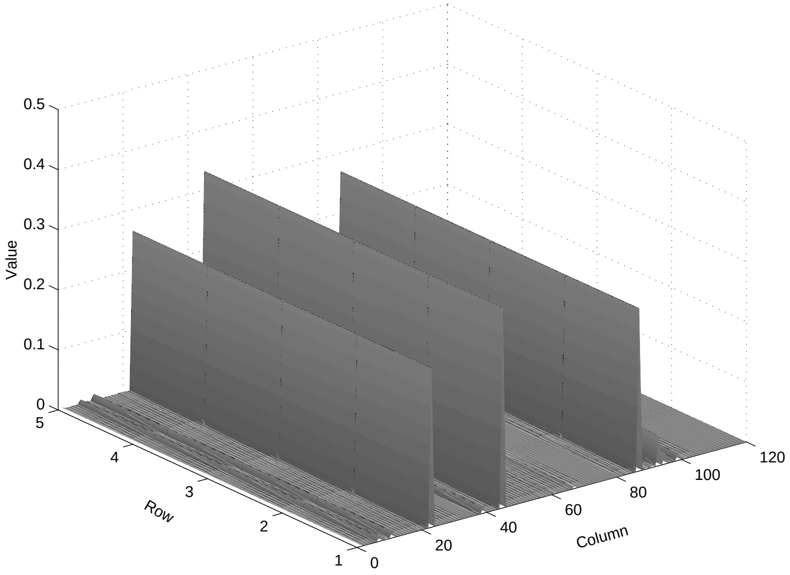

Figure 4.9: Observation matrix (B) for Day 3

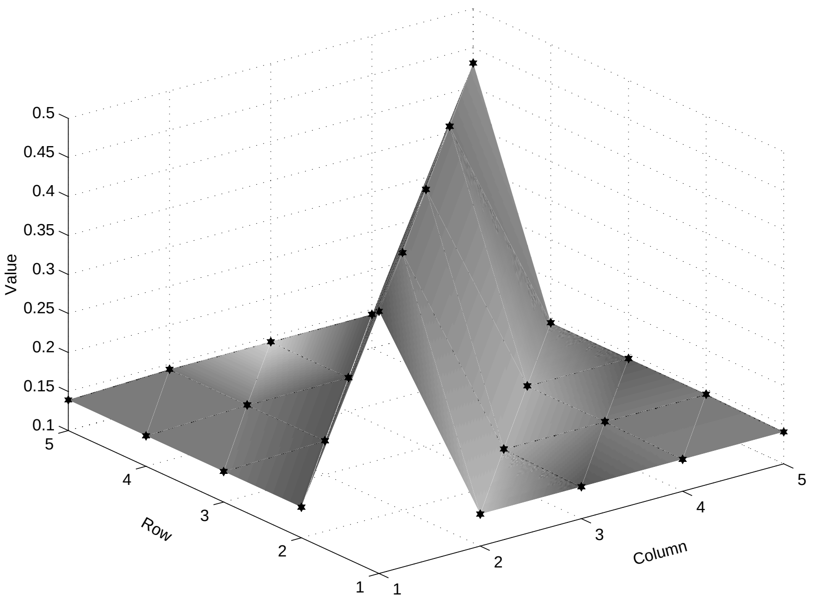

Figure 4.10: Transition matrix (A) for Day 4

## Day 4

Fourth group consists of 8740 sequences of length 100, collected on Friday of the fifth week of collecting data.

The representative model's A matrix is shown in Fig. 4.10.

Also, the representative matrix B for this group is shown in Fig. 4.11.

## Day 5

Fifth group consists of 17058 sequences of length 100, collected on Monday of the fifth week of collecting data.

The representative model's A matrix is shown in Fig. 4.12.

Also, the representative matrix B for this group is shown in Fig. 4.13.

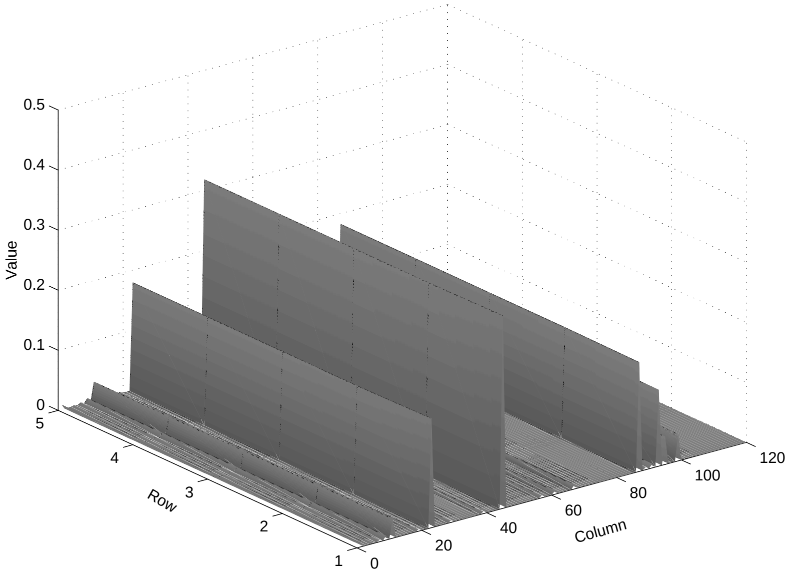

Figure 4.11: Observation matrix (B) for Day 4

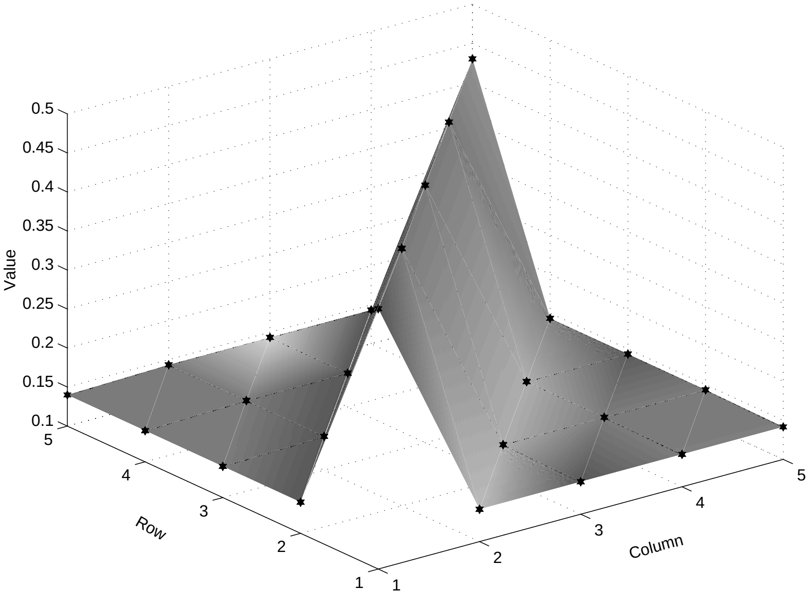

Figure 4.12: Transition matrix (A) for Day 5

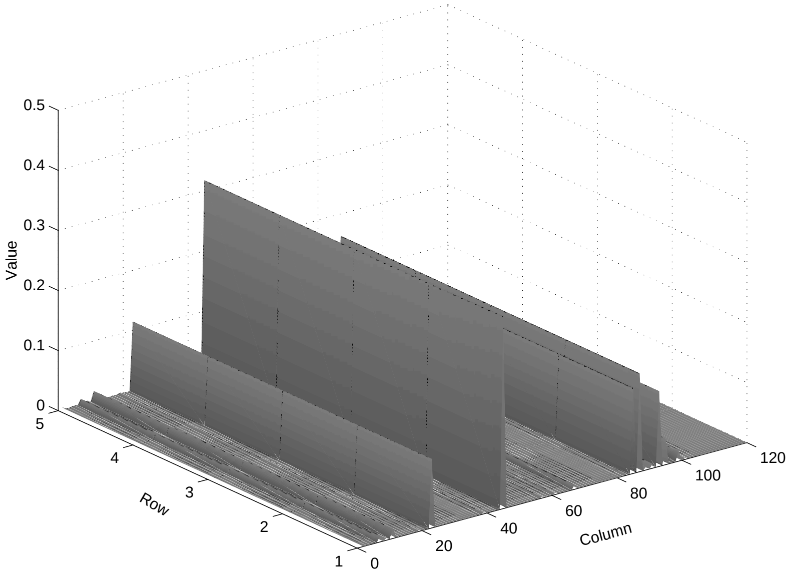

Figure 4.13: Observation matrix (B) for Day 5

## Day 6

Sixth group consists of 11145 sequences of length 100, collected on Tuesday of the fifth week of collecting data.

The representative model's A matrix is shown in Fig. 4.14.

The representative matrix B for this group is shown in Fig. 4.15.

## Day 7

Seventh group consists of 8445 sequences of length 100, collected on Wednesday of the fifth week of collecting data.

The representative model's A matrix is shown in Fig. 4.16.

Also, the representative matrix B for this group is shown in Fig. 4.19.

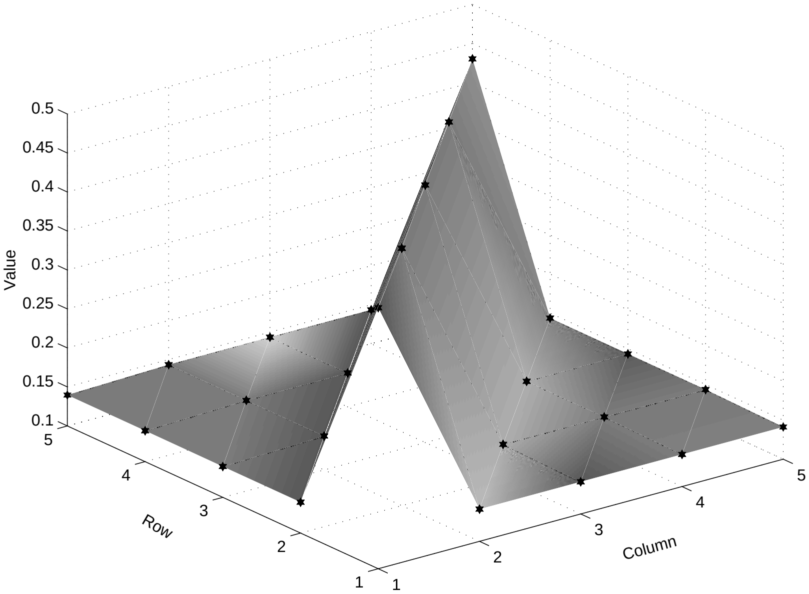

Figure 4.14: Transition matrix (A) for Day 6

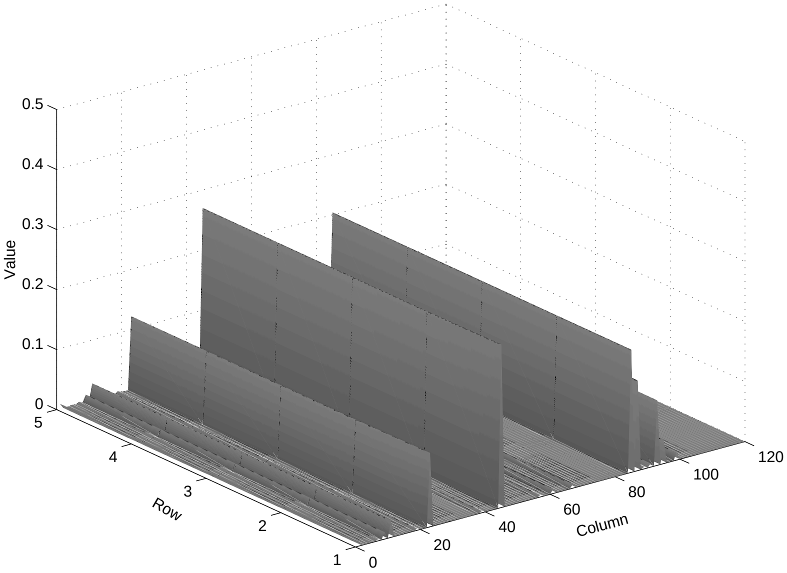

Figure 4.15: Observation matrix (B) for Day 6

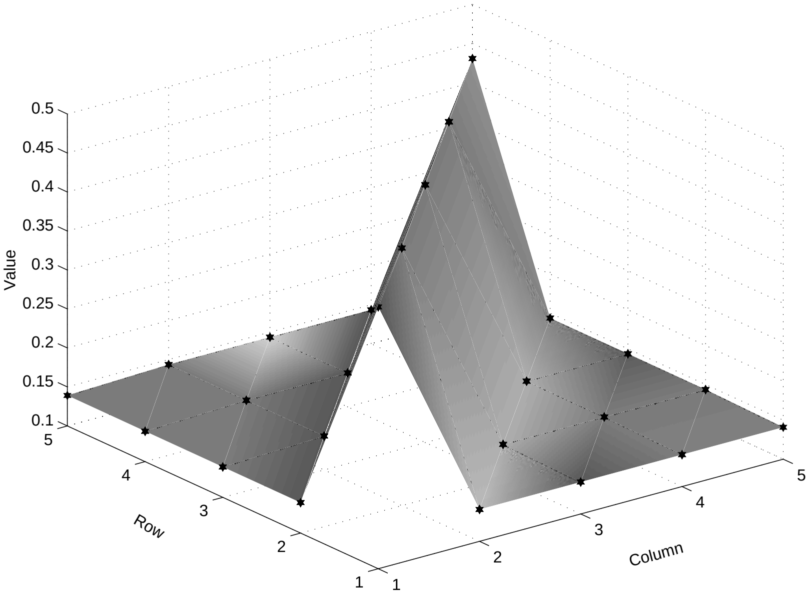

Figure 4.16: Transition matrix (A) for Day 7

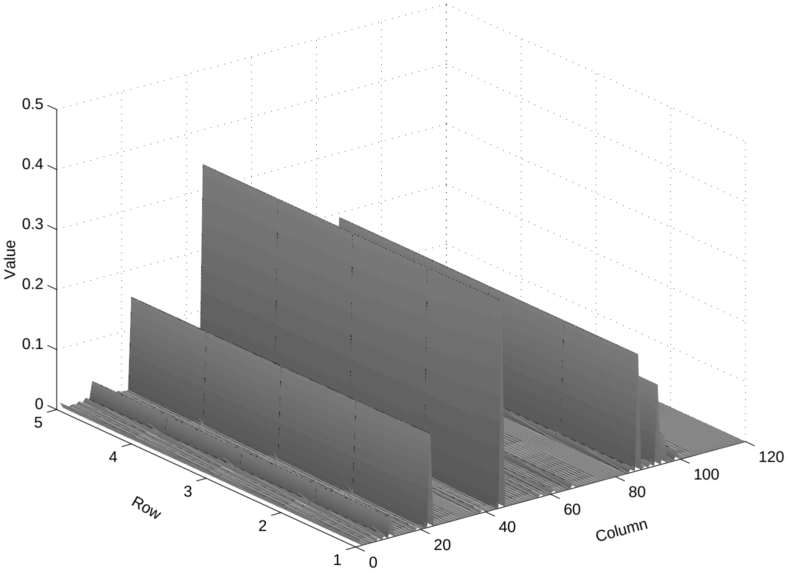

Figure 4.17: Observation matrix (B) for Day 7

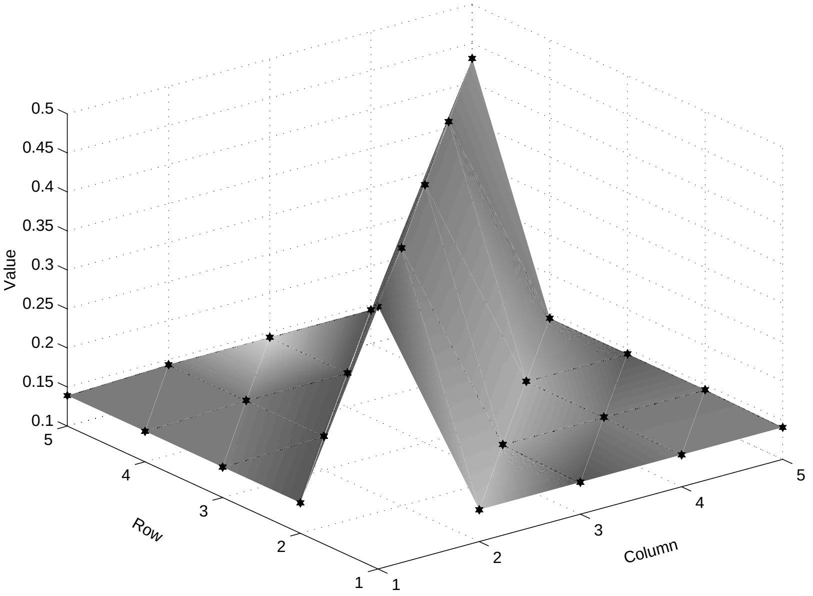

Figure 4.18: Transition matrix (A) for Day 8

### Day 8

Eight group consists of 9143 sequences of length 100, collected on Thursday of the sixth week of collecting data.

The representative model's A matrix is shown in Fig. 4.18.

The representative matrix B for this group is shown in Fig. 4.19.

## 4.1.2 Normal sample sequences

In the verification stage of the proposed algorithm we will need sample sequences of normal behavior. Therefore, it is necessary that we collect them during the initialization phase. It is very subjective what number of sequences should be collected. We decided to keep every thousandth sequence. There is a total of 100722

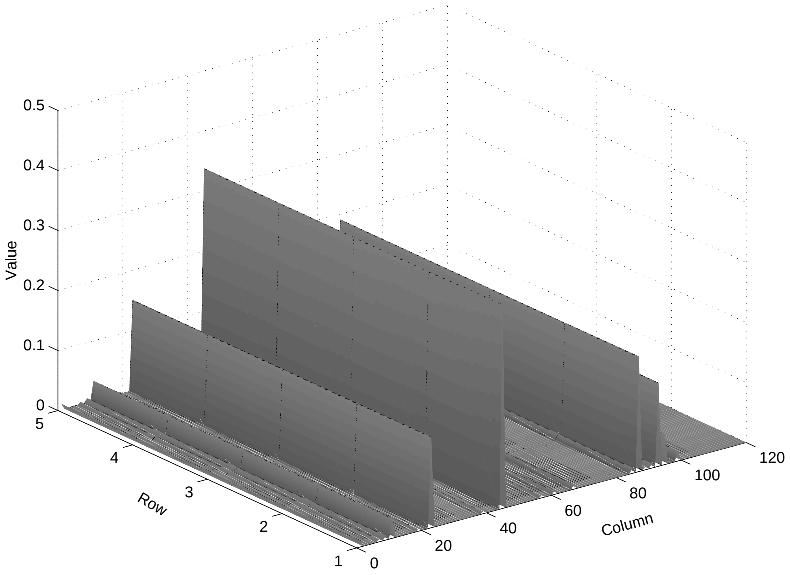

Figure 4.19: Observation matrix (B) for Day 8

normal sequences, which means that in our database of sample normal sequences we will have a hundred of them.

## 4.2 Attack Databases

Since we will have to do verification in the Phase 4, we need to have some attack databases that will help us decide whether a suspicious trace of system calls should be classified as normal behavior or as an attack. Further, it will be used to determine whether, if it turns out to be an attack, it is a variation of an already known attack, or it is a new one.

We will use the idea of attack tree representation [7]. Every attack has a sequence of actions (in the form of states and transitions) that sufficiently describe

| | | | |
|---|---|---|---|
| `eject` | ./ejectexploit ⋯ large header and execution of /usr/bin/eject | E1 | A |
| | large header and execution of /usr/bin/eject ⋯ stat(2) | E2 | B |
| | stat(2) ⋯ /usr/bin/ksh | E3 | C |
| | /usr/bin/ksh ⋯ pipe, fork | E4 | D |
| `ffbconfig` | ./ffbconfig ⋯ large header and execution of /usr/bin/ffbconfig | FF1 | G |
| | large header and execution of /usr/bin/ffbconfig ⋯ /usr/bin/ksh | FF2 | H |
| | /usr/bin/ksh ⋯ pipe, fork | FF3 | D |
| `fdformat` | ./fdformat ⋯ large header and execution of /usr/bin/fdformat | FD1 | I |
| | large header and execution of /usr/bin/fdformat ⋯ /usr/bin/ksh | FD2 | J |
| | /usr/bin/ksh ⋯ pipe, fork | FD3 | D |

Table 4.1: Attack Elementary Instance Database

the attack. We take this attack tree and chunk it into separate branches with states on both ends of the branch. These branches will represent elementary instances of attacks, so we add them to Attack Instance Vocabulary Database. This is done for every known attack. Of course, it can not be assumed that this database will be complete, but we can improve it by adding 'artificial' elementary instances, created by an expert who has knowledge of the programs running on the system to be protected.

The elementary instances for each attack are shown in the Table 4.1 (the last two columns represent the attack instance, the last column is 'global' representation).

| | |
|---|---|
| A | ./ejectexploit $\cdots$ large header and execution of /usr/bin/eject |
| B | large header and execution of /usr/bin/eject $\cdots$ stat(2) |
| C | stat(2) $\cdots$ /usr/bin/ksh |
| D | /usr/bin/ksh $\cdots$ pipe, fork |
| G | ./ffbconfig $\cdots$ large header and execution of /usr/bin/ffbconfig |
| H | large header and execution of /usr/bin/ffbconfig $\cdots$ /usr/bin/ksh |
| I | ./fdformat $\cdots$ large header and execution of /usr/bin/fdformat |
| J | large header and execution of /usr/bin/fdformat $\cdots$ /usr/bin/ksh |
| K | ./ps_exploit $\cdots$ large header and execution of /usr/bin/ps |
| L | large header and execution of /usr/bin/ps $\cdots$ /usr/bin/ksh |
| M | pipe, fork $\cdots$ stat, fork |

Table 4.2: Attack instance vocabulary database

### 4.2.1 Attack Instance Vocabulary Database

Therefore, the Attack Instance Vocabulary Database will consist only of elementary instances from the Table 4.1 above, avoiding double records, as shown in Table 4.2:

### 4.2.2 Attack Sequence Database

Now that we have comprehensive Attack Instance Vocabulary Database, we can make paths for each attack and associate certain weights to each part of a path. This will enable us to calculate the probability that a sequence under consideration

|  |  |
|---|---|
| `eject` | A ⋯ B ⋯ C ⋯ D |
| `ffbconfig` | G ⋯ H ⋯ D |
| `fdformat` | I ⋯ J ⋯ D |

Table 4.3: Attack sequence database

is sufficiently suspicious. For the three attacks, the Attack Sequence Database is shown in Table 4.3:

### 4.2.3 New initialized HMM model

Since our proposed algorithm is also adaptive, we will need an additional, initialized new HMM model that will be trained during testing, in the Phase 2 of parallel testing and training. The purpose of this new model is to capture the current behavior being tested, in order to use it for cross-testing in the verification phase, but, more importantly, to be used for adaptive phase if needed (in case during the execution of our algorithm we decide that the current behavior is normal but not previously seen or abnormal known but not previously seen, or just unknown; we will want to update the corresponding database for future, more accurate use).

At the end of the initialization phase, we have several databases and HMM models for normal and abnormal behavior.

- Normal HMM models
- Normal sample sequences database
- Attack Instance Vocabulary Database

- Attack Sequence Database

- Attack #1 HMM model

- Attack #2 HMM model

- initialized new HMM model

and the goal is to detect and create an HMM model for Attack #3 (unknown) or update the already known models.

# Chapter 5

# Proposed IDS scheme: Phase 2 - Parallel testing and training

Since the goal of our proposed IDS scheme is to detect unknown attacks as well as known, but previously not seen, we will need a way to capture the behavior that is being currently tested so we can use it to update our databases and models. Therefore we are introducing this parallel testing and training phase.

## 5.1 Testing

Testing part of this phase tests sequentially the incoming sequences, one by one. Using the procedure explained in Chapter 2, we first test the sequence against the normal HMM models. Span of the normal HMM models has been adjusted to the data we have, in a way that it covers the virtual circular space not covered by the attack models. Since this is the first discriminative step, and we want to be able to detect all the attacks, known or unknown, we set up a very rigorous threshold for log-likelihood value. Anything that passes the threshold is safe and no further testing is needed. There is also no need to update the normal database, since the sequence being tested already belongs to the normal 'space'. The virtual probabilistic space is shown in Fig. 5.1.

In case we get a negative answer, we suspect that the sequence does not belong to the normal space and we continue testing it against the known attack

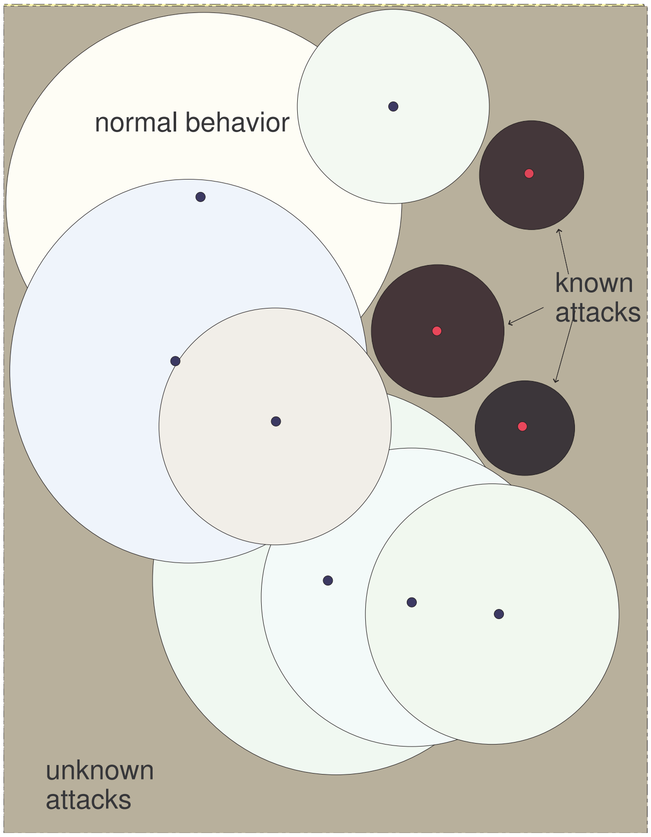

Figure 5.1: Virtual Probabilistic Space

HMM models. Since we have a very rigorous threshold, a lot of sequences will not belong to any of the models - they'll be considered unknown. Those sequences, together with the sequences that gave positive answer to some HMM attack model, are labelled suspicious and passed on to the next phase - logic. We will also have classification here (this is not the final classification; it is here only to give more information to the administrator).

The sequence will be classified as belonging to class `k`, where

$$k = argmax_i Pr(O|\lambda_i), i = 1, 2, 3, N$$

if

$$Pr(O|\lambda_k) > threshold,$$

otherwise,

$$k = U$$

where k=1,2,3,N,U corresponds to Attack #1, Attack #2, Attack #3, Normal behavior and Unknown attack class, respectively.

## 5.2 Training

Along with testing a sequence, we also train the additional, new, initialized HMM with this sequence. This way we make an HMM model of the current sequence that encapsulates system's behavior at that moment and that could be used in the later phases of the algorithm if necessary. This model is being initialized over and over again for every incoming sequence. However, if alarm is raised in Phase 3 of

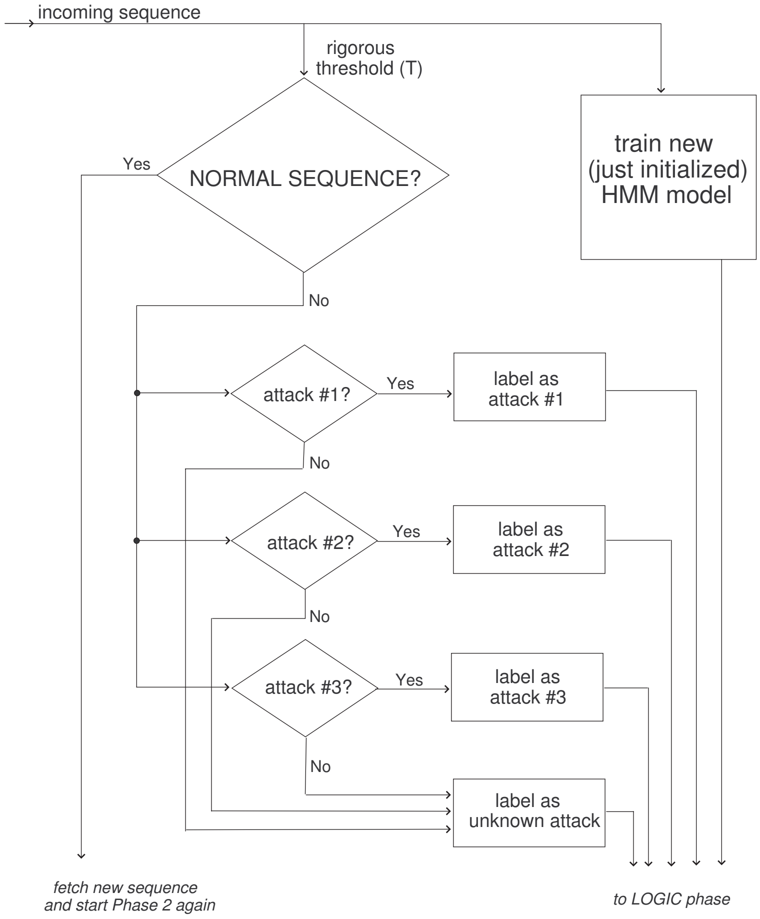

Figure 5.2: Parallel testing and training

the scheme, this model is being used for verification (phase 4) and possible updating of a specific database (phase 5).

The parallel testing and training scheme is shown in Fig. 5.2.

## Chapter 6

## Proposed IDS scheme: Phase 3 - Logic

Logic phase of the algorithm is invoked when a sequence is labelled as suspicious. Since probabilistic testing alone can not give good results we needed to combine different methods in order to reach satisfying results. Since the testing threshold in the previous phase was set as very rigorous, in order to have good detection rate, this fact introduces another type of error: false positives (false alarms). Therefore, from now on the main task will be to differentiate 'good' sequences from 'bad' ones in this group of suspicious traces.

Every sequence consists of 100 system calls. In the logic phase, a sequence can be scanned for elementary sequences listed in Table 4.2. The important part is that this scanning has to capture the sequential nature of elementary sequences, i.e. order is important. Now, it is important to assign probabilities of alarm to every elementary sequence. Having them lined in a specific order, Table 4.2, probabilities are being added up and as soon as a certain threshold value is reached, alarm is raised. Therefore, the probability of the alarm in the third phase will be:

Alarm probability =

$$max\{p_A \cdot I_A + p_B \cdot I_B + p_C \cdot I_C + p_D \cdot I_D,$$

$$p_G \cdot I_G + p_H \cdot I_H + p_D \cdot I_D,$$

$$p_I \cdot I_I + p_J \cdot I_J + p_D \cdot I_D\}$$

The alarm will be raised if this probability exceeds the set threshold:

$$ALARM = (Alarmprobability > alarmthreshold)$$

In case that a sequence passes this logic test, without raising an alarm, the sequence is proclaimed normal and 'sent' to the phase 5 for adaptation, i.e. updating the normal database. In case an alarm is raised, two actions take place: the sequence is passed onto phase 4 - verification and the spare HMM model that was trained with this sequence in phase 2 is kept and sent to the verification phase.

The logic part of the scheme is shown in the figure Fig. 6.1

Since in the real-world applications we might not be able to know in advance all of these specifications of the attacks, we decided to use limited knowledge of information in order to evaluate our IDS scheme. Therefore, for the logic of the alarm we used only two sequential paths of system calls for all the alarms.

$$ALARM =' \texttt{execve}' \cdot I_{\texttt{execve}} \; \& \; ('\texttt{pipe}' \cdot I_{\texttt{pipe}} \mid '\texttt{fork}' \cdot I_{\texttt{fork}}) \qquad (6.1)$$

Hence, note that this phase does not actually contribute to the classification at all, just the detection.

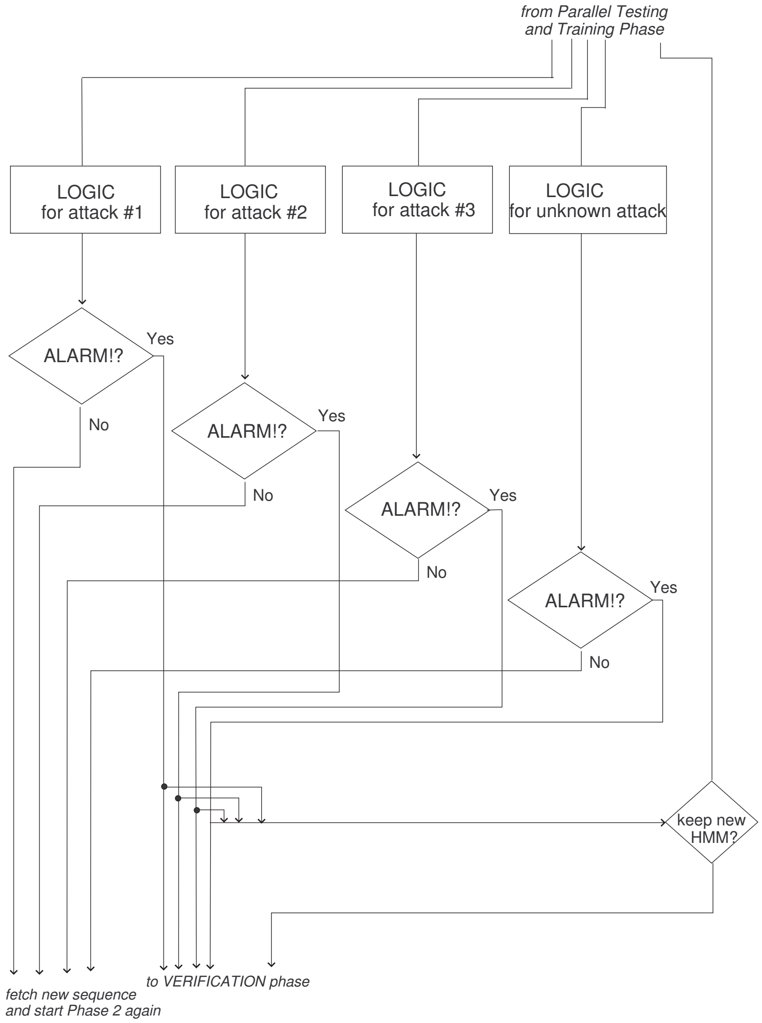

Figure 6.1: Logic

# Chapter 7

## Proposed IDS scheme: Phase 4 - Verification

The verification phase introduces another probabilistic testing, since not every sequence that raises an alarm in the logic phase has to be malicious. Therefore, in this phase we hope to identify more normal sequences that have reached this phase and should not be here. The solution we found lies in cross-testing. It is very well known that log-likelihood estimation is not symmetric. However, not in spite of that fact, but due to it, we are able to extract more information. With this novel approach, we use our new HMM model from phase 2 trained with the current sequence, and test all the normal sample sequences from that database against it. This time, we are not aiming for extremely rigorous threshold value.

Any sequence that passes this cross-test for normal behavior is labelled as normal and sent to the next phase for updating the normal database. If a sequence does not pass this test it is assumed an attack sequence, that needs to be classified. This new HMM model tests sequences of known attacks, and the one with the maximum log-likelihood value is selected as a classification result, but only if the resulting log-likelihood ratio is within the threshold. If it is not, the model is labelled as an unknown attack and passed on to the next phase.

The verification part of this scheme is shown in the figure Fig. 7.1.

The final classification is made in this stage. The sequence will be classified as belonging to class k, where

$$k = argmax_i Pr(O_i|\lambda), i = 1, 2, 3, N$$

if

$$Pr(O_k|\lambda) > threshold,$$

otherwise,

$$k = U$$

where k=1,2,3,N,U corresponds to Attack #1, Attack #2, Attack #3, Normal behavior and Unknown attack class, respectively.

Note that this testing is similar but not the same with the one done in Phase 2. Here we are testing a collection of sequences against one HMM model, while in the Phase 2 we tested (one) incoming sequence against several HMM models.

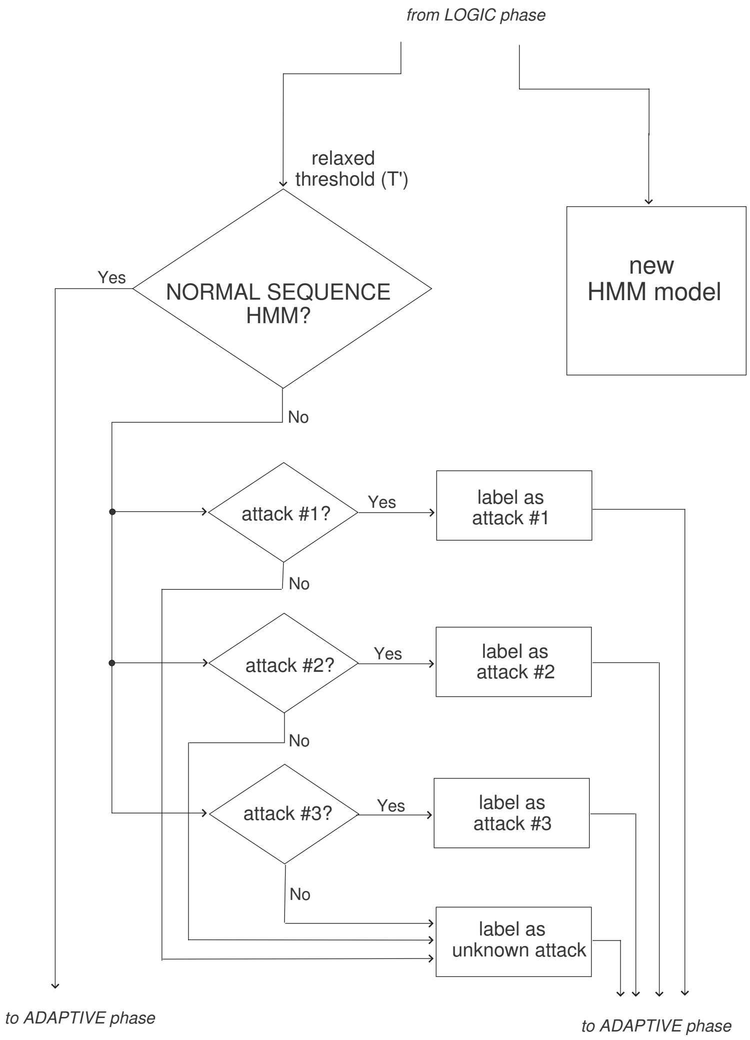

Figure 7.1: Verification

# Chapter 8

## Proposed IDS scheme: Phase 5 - Adaptive phase

Adaptive part of the scheme does exactly that, it updates specific databases with the current (sequence) behavior model passed on from the previous phases.

If a sequence that went through the algorithm was labelled as a known attack (but that sequence was not previously seen, of course), then the model created in phase 2 and verified in phase 4 is simply added to the corresponding database.

If a sequence was labelled as unknown attack, the model for it is stored in the unknown attack database.

However, if a sequence passed the test, and was labelled as normal, the model is not just simply added to the normal database. Since the normal database consists of only 8 HMM models, adding a model for each not previously seen normal sequence would significantly change the database, and hence enable adversary to actually train our IDS system and then attack it without a risk of being caught. Therefore, a new HMM does participate in the normal database, but it's added proportionally to it's contribution. In other words, if we decide to add a new HMM model to an HMM model comprised of 10000 (averaged) different HMM's, it's contribution to the representative model will be 1/10001 compared to 10000/10001 contribution of the existing model. This way, we protect our IDS from adversary training.

The complete scheme incorporates all five phases and is shown in Fig. 8.2.

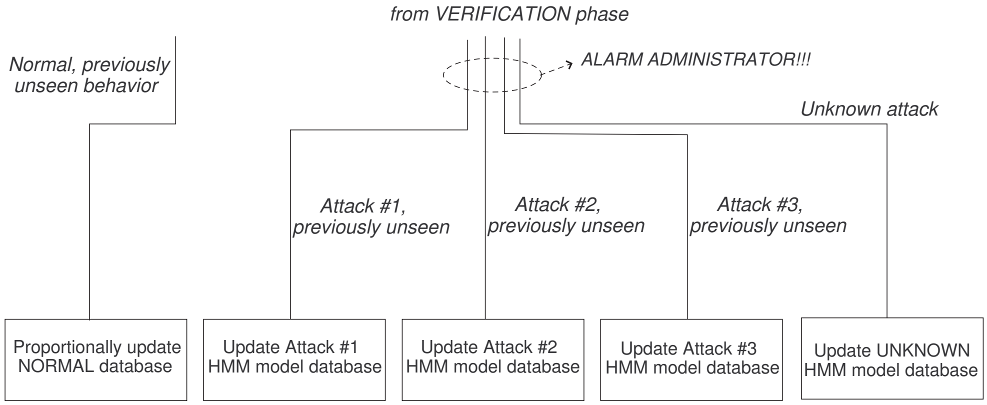
from VERIFICATION phase
ALARM ADMINISTRATOR!!!
Normal, previously unseen behavior
Unknown attack
Attack #1, previously unseen
Attack #2, previously unseen
Attack #3, previously unseen
Proportionally update NORMAL database
Update Attack #1 HMM model database
Update Attack #2 HMM model database
Update Attack #3 HMM model database
Update UNKNOWN HMM model database

Figure 8.1: Adaptive phase

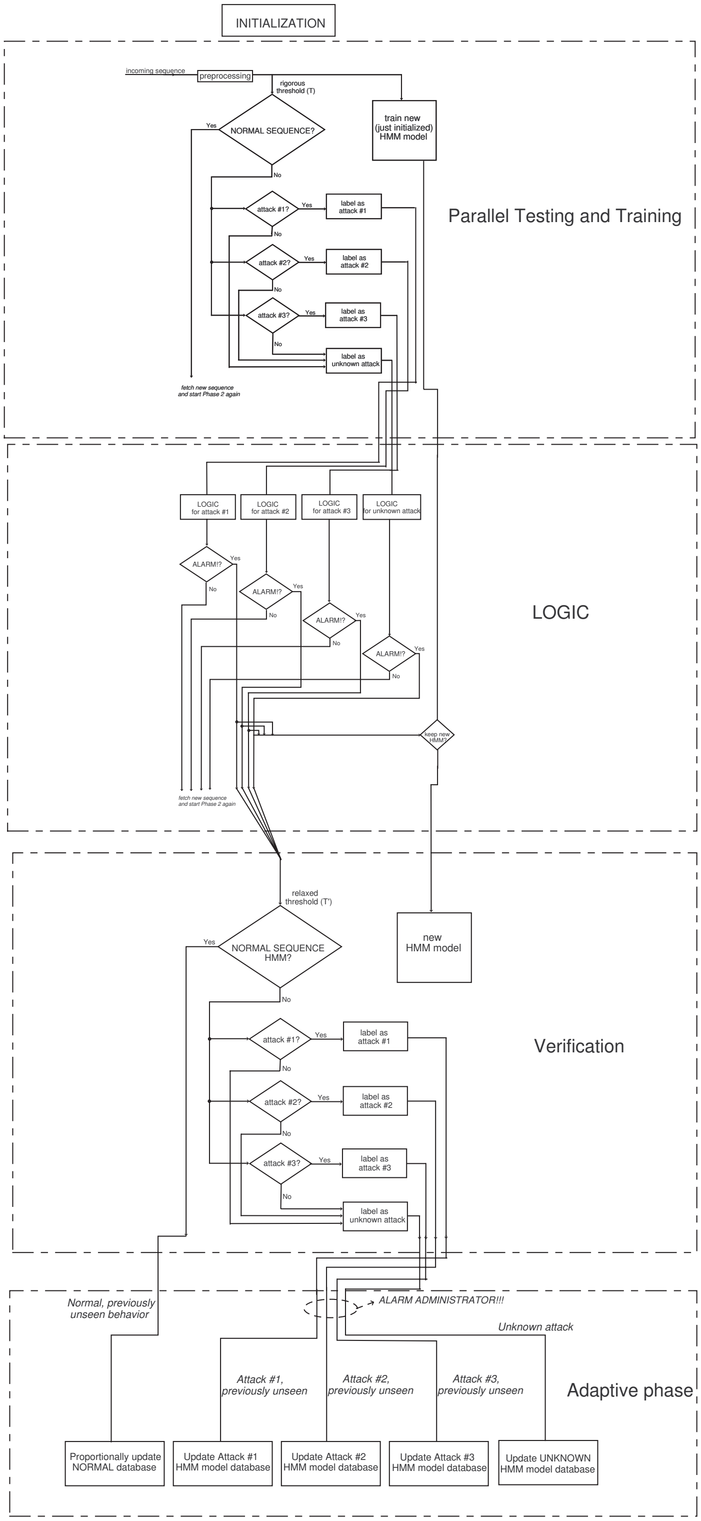

Figure 8.2: Complete scheme

# Chapter 9

# Results

## 9.1 Known attack detection and classification

Under 'known attack', we mean a type of an attack that has happened before and we have some knowledge about it. We tested FFB attack as 'known attack', i.e. we had previous instances of that type of attack happen before, so we do have some idea how it should look like. In our data there were 9 instances of FFB attack, some during the same day. We tested each one of them as a new instance of a known attack. Of course, we had a database of several models for that FFB attack, but also databases for other two attacks, FORMAT and EJECT. Also, we had our normal models database. In all the cases, the attack was detected, and, in this case, perfectly classified.

When we tested FORMAT attack as a known attack, we used 3 instances of the attack. Those instances were happened on Day 1, Day 4 and Day 5 of data collection **??**. Due to the good solution of our scheme we have a perfect detection, but, since there is such a few data, it was not possible to appropriately 'cover' the space of the FORMAT attack, therefore we have misclassifications. One instance of the FORMAT attack was classified correctly, one was misclassified as an FFB attack and one was labelled as unknown (which we prefer over labelling it as a wrong type of known attack).

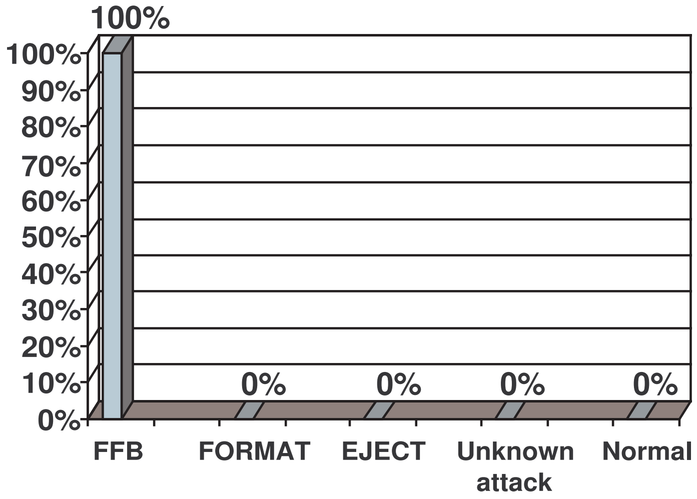

Figure 9.1: Known attack #1 - FFB

There was 7 instances of EJECT attack, two of them were in the Day 3, and the rest of them in Day 4 through Day 8. However, it seems that each instance of this attack was pretty specific, so there were some misclassifications: only one instance was classified correctly, two were classified as FFB attack and the rest were labelled unknown.

Overall, it seems that the FFB attack is the most general one, and the other two tend to be misclassified as FFB. However, note that this is the worst case scenario for our scheme, since all three attacks belong to the same type of attacks - buffer overflows. In case we had more different attacks, the results would be much better. Also, we used only limited knowledge of the attacks. If we include more information, the results will only get better. Our objective is to show that we can do very well using as limited information as possible.

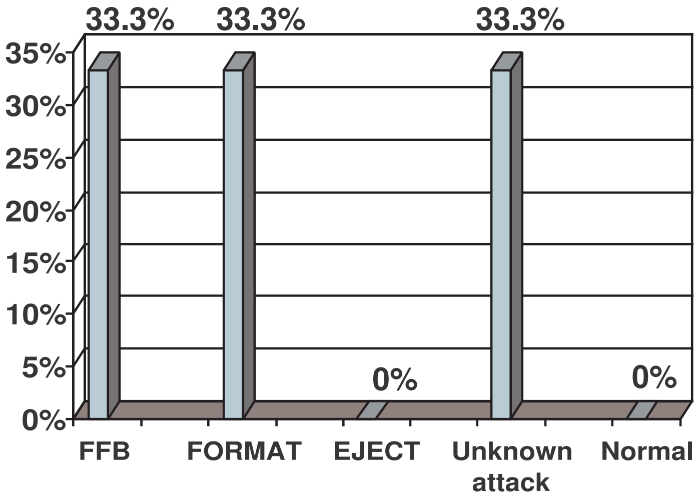

Figure 9.2: Known attack #2 - FORMAT

## 9.2 Unknown attack detection and classification

In this experiment, we assumed that out of our three types of attacks, we have seen only two before, i.e. we have no prior knowledge of the third attack whatsoever.

First, we tested FFB as an unknown attack. The log-likelihood ratios were compared with (other two) known attacks and normal behavior, and if a threshold was not met, the sequence is labelled unknown. All of the 9 sequences of the unknown attack (FFB) were detected and classified as such. Also, all the models of those sequences are saved and stored for future use (as explained in Adaptive phase of the scheme).

Secondly, we tested FORMAT attack as unknown (we had no models of this type of attack stored). For the reasons explained earlier, the obvious generality of FFB behavior, we had some misclassifications: two instances were classified as FFB attack and one was classified correctly as unknown.

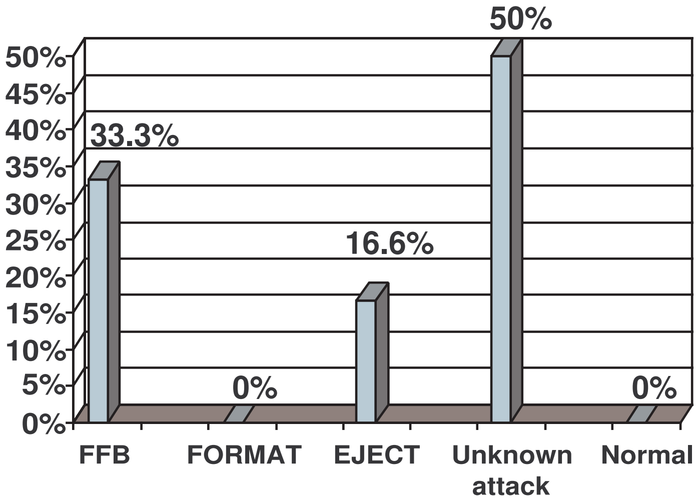

Figure 9.3: Known attack #3 - EJECT

Testing EJECT attack as unknown gave better results. Only two instances of this attack were misclassified as FFB, and other four were labelled correctly as unknown attack.

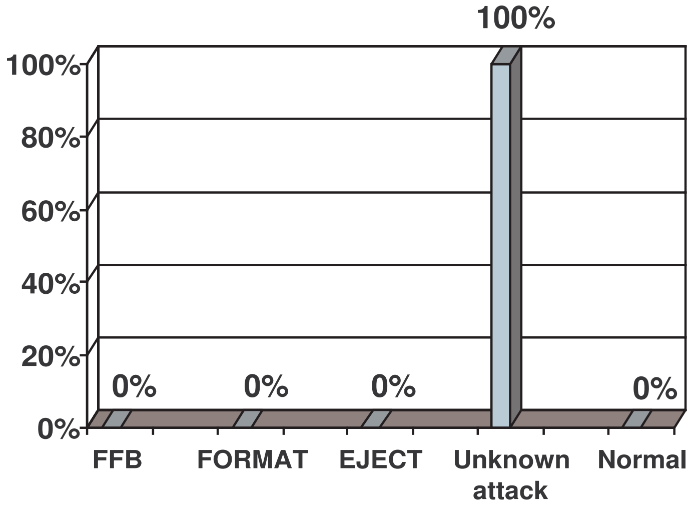

Figure 9.4: Unknown attack #1 - FFB

## 9.3 Normal behavior classification - lowering false alarm rate

Since the principle idea of our proposed scheme is to set up desired detection rate and then filter out false positives (false alarm), here we have results for testing normal behavior of the system. Ideal result would be that all the sequences are classified as normal at the end. However, that is not the case, we do have false positives; however, the algorithm we designed filters them out through stages, so the end false positive rate is very promising.

Test data in Day 1 consists of 7330 sequences of length 100. Each one is tested with our algorithm and the results are shown in Fig. 9.7. After phase 2, Parallel Testing and Training phase, we had 428 suspicious sequences. All of those were sent through to Logic phase, where they were scanned for specific pairs of system calls. Only the ones who satisfied our condition for malicious behavior (in this case 50)

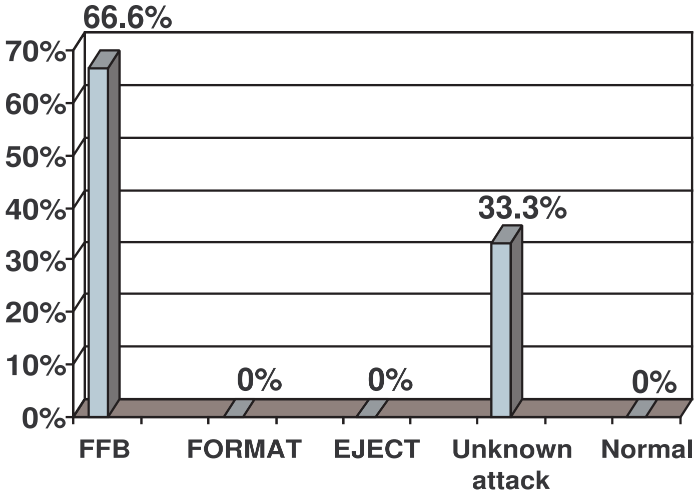

Figure 9.5: Unknown attack #2 - FORMAT

were passed onto the next phase of the algorithm for verification. At the end, we had 9 sequences that were classified as anomalous. The false alarm rate for this set of data was 0.1228%.

Normal behavior data from Day 2 consists of 19302 sequences of length 100. Following the same procedure explained above for Day 1 test data, after the Parallel Testing and Training phase we had 954 suspicious sequences there were sent to the Logic phase. After the Logic phase, we had 121 sequences that satisfied condition we imposed for malicious sequences which we sent on to the next phase for verification. There were 31 sequences classified as anomalous, which makes false alarm rate 0.1604%. The results are shown in Fig. 9.8.

Test data of Day 3 consists of 19529 sequences of length 100. Going by the algorithm, after the phase of Parallel Testing and Training we had 749 suspicious sequences that had to be passed on the the next, Logic phase. Only 72 sequences

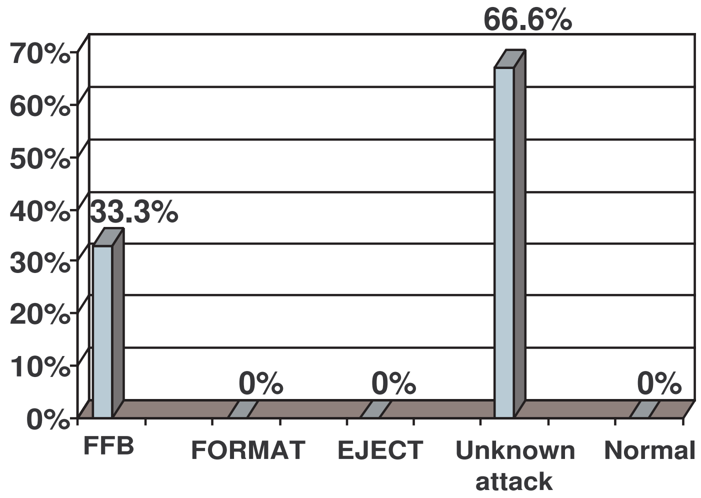

Figure 9.6: Unknown attack #3 - EJECT

came out of the Logic phase satisfying the conditions for malicious behavior and they were passed on to the next stage for verification. There were only 5 sequences classified as anomalous after the execution of the whole algorithm, which makes the false alarm rate very low for this test data - 0.0256%, which is shown in Fig. 9.9.

Test data in Day 4 consists of 8740 sequences of length 100. The sequences are sent through the algorithm and the results are shown in Fig. 9.10. After phase 2, Parallel Testing and Training phase, we had 918 suspicious sequences. All of those were sent through to Logic phase. Only the ones who satisfied our condition for malicious behavior ( 54 in this case) were passed onto the next phase of the algorithm for verification. At the end, we had 5 sequences that were classified as anomalous. The false alarm rate for this set of data was 0.0572%.

There are 17058 normal behavior sequences from Day 5. Following the same procedure as above, after the Parallel Testing and Training phase we had 1055

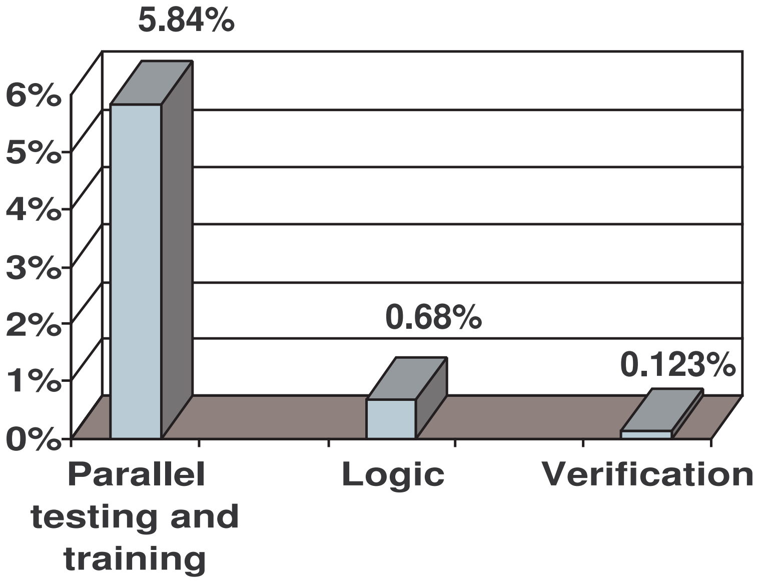

Figure 9.7: False positive rate through stages - Day 1

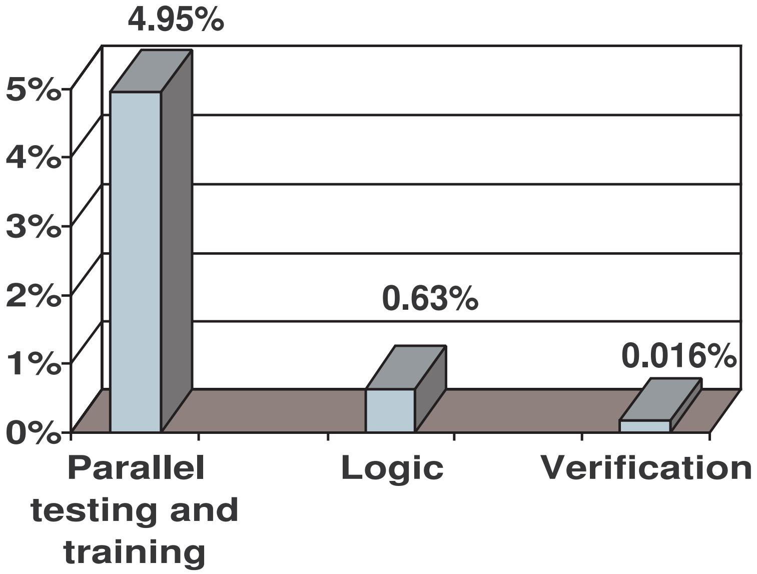

Figure 9.8: False positive rate through stages - Day2

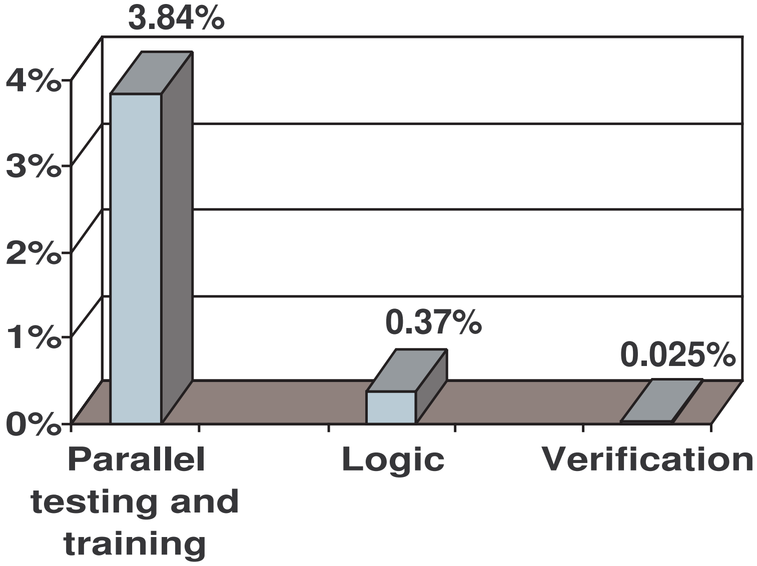

Figure 9.9: False positive rate through stages - Day 3

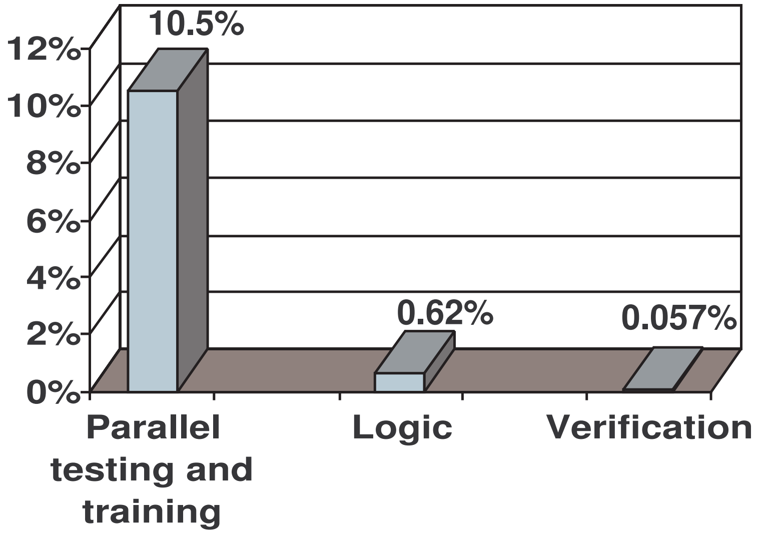

Figure 9.10: False positive rate through stages - Day 4

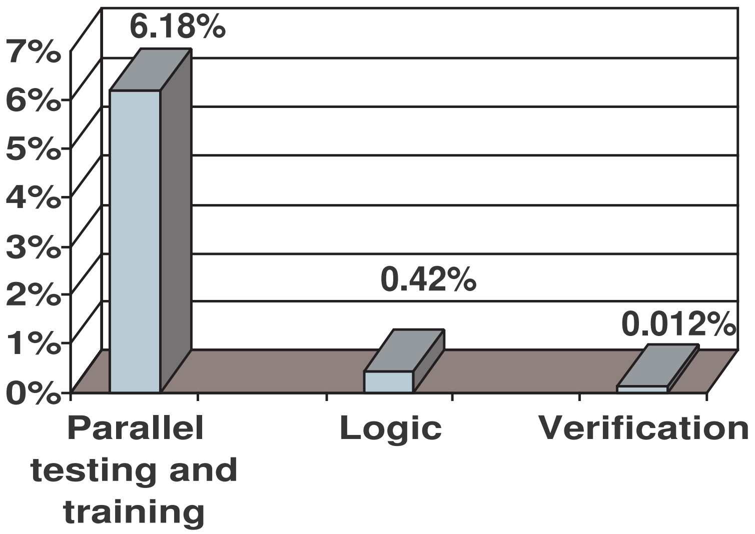

Figure 9.11: False positive rate through stages - Day 5

suspicious sequences there were sent to the Logic phase. After the Logic phase, we had 71 sequences that satisfied condition we imposed for malicious sequences which we sent on to the next phase for verification. There were only 2 sequences classified as anomalous, which makes false alarm rate very low - 0.0117%. The results are shown in Fig. 9.11.

Test data of Day 6 consists of 11145 sequences of length 100. After the phase of Parallel Testing and Training we had 778 suspicious sequences that had to be passed on the the next, Logic phase. Out of those, 96 sequences came out of the Logic phase satisfying the conditions for malicious behavior and they were passed on to the next stage for verification. There were 23 sequences classified as anomalous after the execution of the whole algorithm, which makes the false alarm rate 0.2064%, which is the maximum value of the false alarm rate over all days of normal behavior; results are shown in Fig. 9.12.

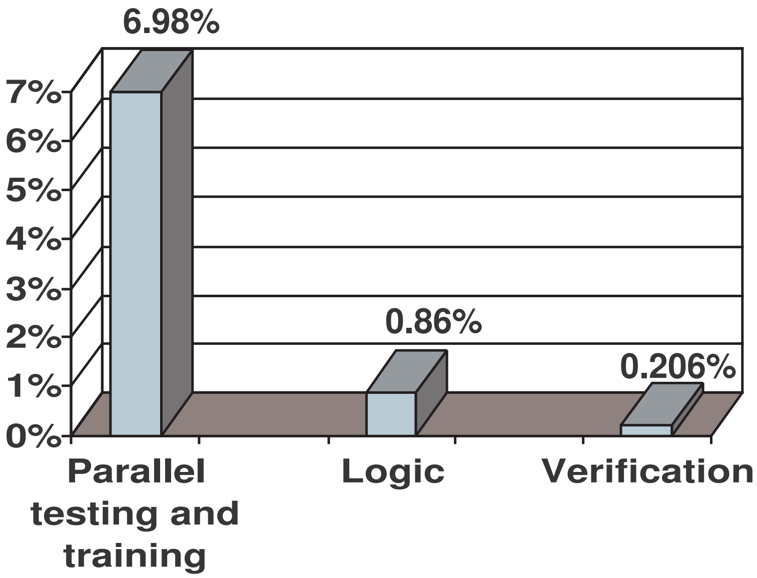

Figure 9.12: False positive rate through stages - Day 6

Test data in Day 7 consists of 8445 sequences of length 100. The sequences are sent through the algorithm and the results are shown in Fig. 9.13. After phase 2, Parallel Testing and Training phase, we had 535 suspicious sequences. All of those were sent through to Logic phase. Only the ones who satisfied our condition for malicious behavior (58 in this case) were passed onto the next phase of the algorithm for verification. At the end, we had total of 7 sequences that were classified as anomalous. The false alarm rate for this set of data was 0.0829%.

There are 9143 normal behavior sequences from Day 8. Following the same procedure as above, after the Parallel Testing and Training phase we had 632 suspicious sequences there were sent to the Logic phase. After the Logic phase, we had 56 sequences that satisfied condition we imposed for malicious sequences which we sent on to the next phase for verification. There were 6 sequences classified as anomalous,

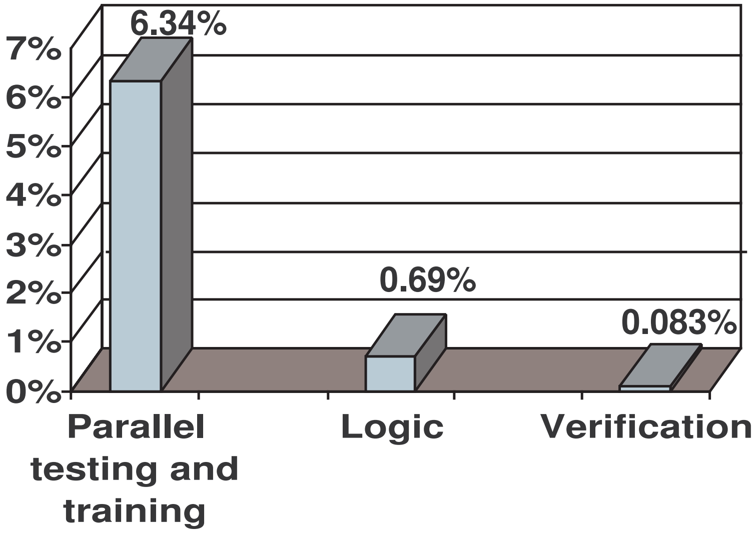

Figure 9.13: False positive rate through stages - Day 7

which makes false alarm rate 0.0656%. The results are shown in Fig. 9.14.

All of the above results show the tendency of radically improving (decreasing) false alarm rate. This is very important since while it is relatively easy to improve detection rate (for example by adding firewalls), it is not easy to get a good (low) false positive rate. Comparative analysis of false alarm rate reduction through stages for all 8 partitions of our normal behavior data is shown in Fig. 9.15 and it shows the effect of our algorithm on false alarm rate.

Average false alarm rate, which is one of the main performance characteristics of our algorithm, is shown in Fig. 9.16.

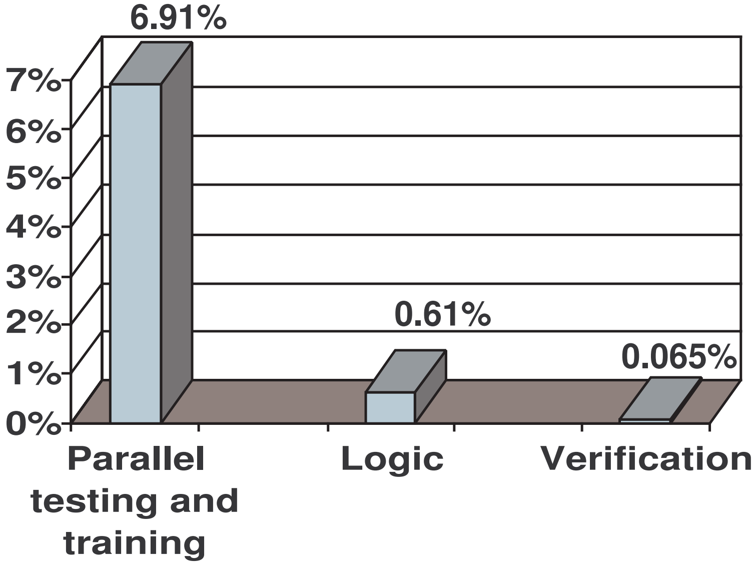

Figure 9.14: False positive rate through stages - Day 8

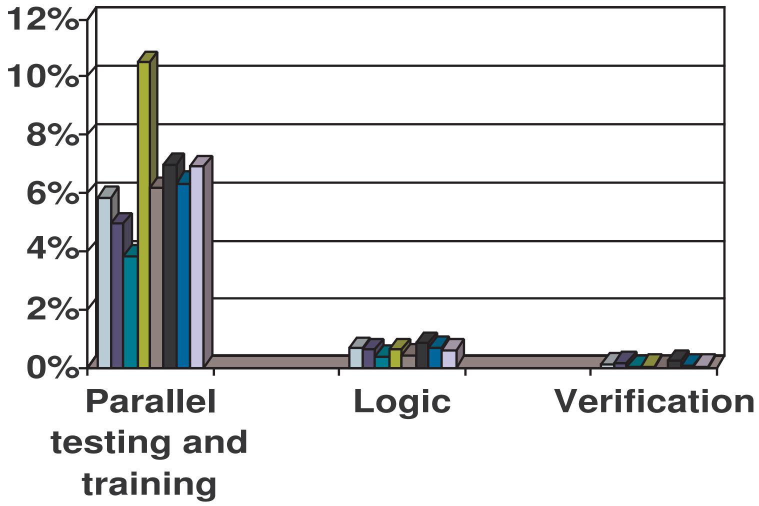

Figure 9.15: False positive rate for all normal data through stages

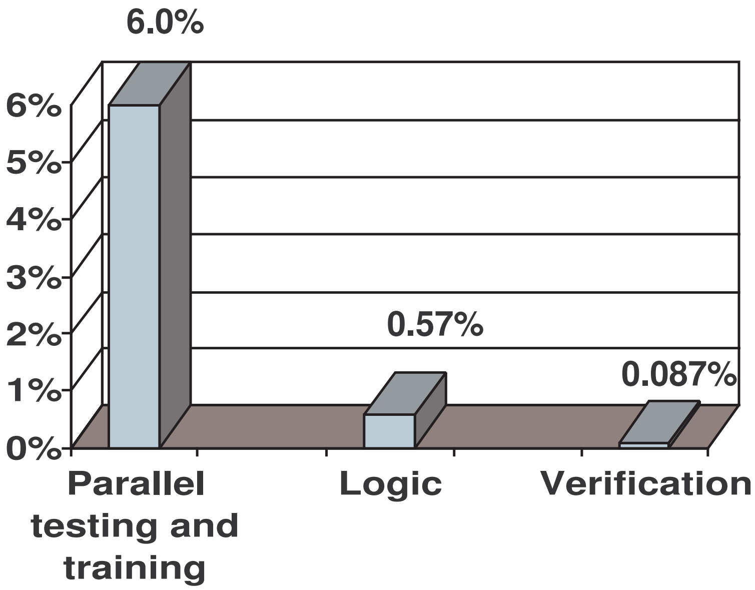

Figure 9.16: Average false positive rate through stages

# Chapter 10

# Conclusions

The goal of our proposed IDS solution was to find a way of implementing a good trade-off between misuse and anomaly detection, resulting in perfect detection and low false positive rate. The main component of it is detection and classification of unknown network attacks.

Our scheme produces new models of previously unknown attacks as well as of known attacks which have not been seen before in that form. The models are there for administrator to look at them and confirm the nature of the possible attack and classification. It relieves the administrator from looking through huge lists of audit logs trying to find where something went wrong. Another advantage of the scheme is that if any other information was used (like raw BSM data in this case), no misclassification would be made (for known and unknown attack testing).

We can also observe that FFB attack seems to be more general than FORMAT and EJECT, therefore there is a tendency to misclassify the other two as FFB.

Our main idea is to set detection high enough - in our case it is 100%, and then filter out false positives through stages. The false positive rate we ended up with seems to be very good - around 0.08%.

Another advantage of our scheme is that in our simulations all the attacks belong to one group of attacks - buffer overflow attacks - which is a worst case scenario, attacks tend to 'look alike'. If we had other group of attacks to detect and classify, we expect the results to be much better, since the models would differ more among themselves.

We believe that finer tuning of probabilistic boundaries and better expert (administrator) knowledge give room for significant improvement.

The drawback of this scheme is that we need extensive expert knowledge in order to set up the databases correctly, and to include all possible dangerous events; also, we need to be able to define the log-likelihood boundaries in the probabilistic space as well as possible.

Our goal is to show that it is possible to detect and classify attack using the mixture of probabilistic testing and logic, when we use only limited number of parameters. In this case we used only system calls. In the real-world applications, we might not know all the parameters attacker will use to compromise a system, especially in the case of an unknown attack.

# BIBLIOGRAPHY

[1] L. R. Rabiner, *A tutorial on Hidden Markov Models and selected applications in speech recognition*, Proc. IEEE, 77(2), 257-286.

[2] R. Dugad, U.B. Desai *A tutorial on Hidden Markov Models*, Tech. Rep. SPANN-96.1, 1996.

[3] G.D.Forney Jr., *The Viterbi Algorithm*, Proc. IEEE, vol.61, no.3, pp.263-278, March 1973.

[4] IST Group at LL, 1998 DARPA Intrusion Detection Evaluation Data Set Overview, project described at http://www.ll.mit.edu/IST/ideval/

[5] K. Kendall, *A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems*, M.Eng. Thesis, MIT Department of Electrical Engineering and Computer Science, June 1999.

[6] R.P. Lippmann, R.K. Cunningham, S.E. Webster, I. Graf, D. Fried, *Using Bottleneck Verification to Find Novel New Computer Attacks with a Low False Alarm Rate*, Unpublished Technical Report, 1999.

[7] B. Schneier, *Attack Trees*, Dr. Dobb's Journal, December 1999.

[8] MIT toolbox for HMM , http://www.ai.mit.edu/ murphyk/Software/HMM/hmm.html

[9] SecurityFocus, Intrusion Detection Systems Terminology, described at httP://www.securityfocus.com/infocus/1213