Detection and Classification of Network Intrusions Using Hidden Markov Models [1]

Svetlana Radosavac
Electrical and Computer Engineering Department
and the Institute for Systems Research
University of Maryland
College Park, MD 20742
e-mail: svetlana@isr.umd.edu

John S. Baras
Electrical and Computer Engineering Department
and the Institute for Systems Research
University of Maryland
College Park, MD 20742
e-mail: baras@isr.umd.edu

Abstract — **This paper demonstrates that it is possible to model attacks with a low number of states and classify them using Hidden Markov Models with very low False Alarm rate and very few False Negatives. We also show that the models developed can be used for both detection and classification. We put emphasis on detection and classification of network intrusions and attacks using Hidden Markov Models and training on anomalous sequences. We test several algorithms, apply different rules for classification and evaluate the relative performance of these. Several of the attack examples presented exploit buffer overflow vulnerabilities, due to availability of data for such attacks. We emphasize that the purpose of our algorithms is not only the detection and classification of buffer overflows; they are designed for detecting and classifying a broad range of attacks.**

I. Introduction

With the increased use of networked computers for critical systems, computer network security is attracting increasing attention and network intrusions have become a significant threat in recent years. The number of intrusions is dramatically increasing and so are the costs associated with them. With an increased understanding of how systems work intruders have become more skilled at determining weaknesses in systems and exploiting them to obtain increased privileges and at the same time the knowledge necessary to carry out an attack is decreasing, resulting in a rapid increase of attempted attacks on various systems and networks.

The primary sources of information for intrusion detection systems are network activity and system activity. Network-based systems look for specific patterns in network traffic and host-based systems look for those patterns in log files. We can group intrusion detection systems into two classes: *anomaly detection systems* and *signature based systems*. Anomaly detection systems attempt to model the usual or acceptable behavior. They have high false positive rates and usually have detection delay. Misuse detection refers to intrusions that follow well-defined patterns of attack that exploit weaknesses in the system. Misuse detection techniques have high detection rate for already known attacks but even the slightest variation of an attack signature may cause the system not to recognize the attack.

Attack detection can be done by observing sequences of system calls. In [5] Hofmeyr uses the rule of r contiguous bits on sequences of system calls and in [6] Warrender *et al* models normal behavior of data sets using stide, t-stide, RIPPER and HMMs. Her experiments proved that HMMs have the best performance, but when normal behavior is modelled training of HMMs is not time efficient. Sekar *et al* [7] introduced a fast-automation based method for detecting program behaviors. Irregular behavior is flagged as potentially intrusive. All those methods are based on anomaly detection. The approach presented in this paper is based on misuse detection and the training sequences are represented in the form of short sequences of system calls.

We are aware of the existence of tools for static code checking and their advantages over Intrusion Detection Systems [8, 9]. If implemented correctly and in all programs all buffer overflows would be detected. However, they also have some disadvantages (high false alarms, high false negatives, cannot detect all network attacks etc.) and are not applied in all programs.

The paper is organized as follows. In section II we present 2 methods of attack modelling and present models for several attacks using one of the representations. In section III we present the problem of detection and classification using Hidden Markov Models. In section IV we present the algorithm for detection and classification. In section V we state the main results and in section VI we present the conclusions and possible extensions of the stated work.

II. Attack representation and modelling

The most comprehensive way of representing an attack is in the form of an *attack tree*. An attack tree [2] is a Directed Acyclic Graph (DAG) with a set of nodes and associated sets of system calls for those nodes. Attack trees provide a formal, methodical way of describing the security of systems, based on various attacks. They give a description of possible sets of subactions that an attacker needs to fulfill in order to achieve a goal and bring the network to an unsafe state. The individual goals can be achieved as sets of `OR` and `AND` sets of actions and subactions.

However, in order to check every possible subset of attacks we need to run the check exponential number of times to the number of attacks and another difficulty is represented in the form of interleaved attacks. Therefore, instead of representing each attack with the subset of all possible actions we use the Finite State Machine (FSM) approach where each attack is represented with a very low number of states and we show the method is highly efficient in both detection and classification of attacks.

The attacks studied in this paper are: `eject, ps, ffbconfig` and `fdformat` and all of them belong to the class

[1] Research partially supported by the U.S. Army Research Office under grant No DAAD19-01-1-0494

of User to Root exploits. User to Root exploits belong to the class of attacks where the attacker gains the access to a normal user account on the system and by exploiting some vulnerability obtains the root access. The model for `ftp-write` attack was also developed but due to the lack of data the tests could not be performed using the developed model. Certain regularities were captured in behavior of exploited programs by comparing them against normal instances of those programs, other instances of attacks detected at different periods of time and by searching for certain events in the behavior of a program that were expected to happen by using the knowledge about the mechanism of those attacks and their goals.

The most common User to Root attack is buffer overflow attack, which enables the attacker to run personal code on a compromised machine once the boundary of a buffer has been exceeded, giving him the privileges of the overflowed program (which in most cases is root). This type of attacks usually tries to execute a shell with the application's owner privileges. Some examples of those attacks are `eject, ffbconfig, fdformat, loadmodule, perl, ps, xterm` etc.

The studied examples show that each attack is characterized with a very simple distinguishing sequence of system calls and accompanying parameters (like size, PID, path, etc.), which can be used for recognition and identification of different attacks.

Due to unavailability of data for other types of attacks we present the results only for several buffer overflow attacks.

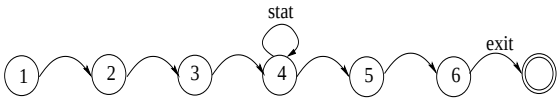
A Eject attack - U2R

The eject attack exploits a buffer overflow vulnerability in eject program. If exploited, this vulnerability can be used to gain root access on attacked systems.

We extracted numerous instances of both normal and anomalous sequences of eject program and noted regularities in program behavior. We also examined one instance of *stealthy* eject attack and noted the same regularities in program behavior as in clear instances of that attack. That can be explained with the fact that the stealthy part of the attack was performed in the login part and file transport to the exploited machine (i.e. encrypted file, scheduled exploit execution etc.) The attack traces consist of hundreds or thousands lines of system calls. However, there are only a couple of system call sequences that are sufficient to completely define the attack. Another identifying string that characterizes the eject exploit is `usr/bin/eject` or existence of string `./ejectexploit` or `./eject`. We will see that similar sequences characterize the other buffer overflow exploits. Buffer overflow is detected by observing large value of header and a sequence of symbols such as ! @ # $ % & . This may not be the case at all times because a skilled attacker may use a combination of letters and non-repeating patterns instead of repeating a short sequence of patterns.

The diagram of `eject` attack is presented in figure 1.

Another serious problem in attack detection is *globbing* which is used to avoid detection of suspicious commands. For example, instead of typing command `/bin/cat/etc/passwd` the attacker will use `/[r,s,t,b]?[l,w,n,m]/[c,d]?t /?t[c,d,e]/*a?s*`. The shell will replace the glob characters and will find that the only valid match for this string is `/bin/cat/etc/passwd`. None of the attacks from the MIT data set [1] had examples of globbing.



1. Start
2. Eject program starts
3. Buffer overflow finished
4. Polling devices
5. Shell executed
6. Child process created

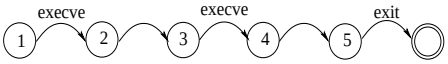
1−>2: ./ejectexploit
2−>3: large header, execution of \usr\bin\ejec
3−>4: stat(2)
4−>5: /usr/bin/ksh
5−>6: pipe, fork

Figure 1: Eject attack.

B Ffbconfig - U2R

This attack is another example of a buffer overflow attack that has signature very similar to the signature of `eject` attack. There is one state less in the equivalent representation than in the state diagram constructed for eject attack.

The equivalent state diagram is represented in Figure 2.



1. Start
2. Ffbconfig program starts
3. Buffer overflow finished
4. Shell executed
5. Child process created

1−>2: ./ffbconfig
2−>3: large header, execution of /usr/bin/ffbconfig
3−>4: shell execution: /usr/bin/ks
4−>5: pipe, fork

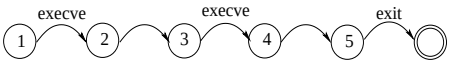
Figure 2: Ffbconfig attack.

C Fdformat - U2R

`Fdformat` attack is another example of buffer overflow attack. The Fdformat attack exploits a buffer overflow in the 'fdformat' program distributed with Solaris 2.5.

This exploit is almost identical to `ffbconfig` attack. The only differences are in the path of the attack which is in this case `/usr/bin/fdformat` and in the file that is executed (if it is named by the attack) `./formatexploit` or `./format`.

The equivalent state diagram is represented on Figure 3



1. Start
2. Fdformat program starts, loading libraries
3. Buffer overflow finished
4. Shell executed
5. Child process created

1−>2: ./fdformat
2−>3: large header and execution of/usr/bin/fdformat
3−>4: shell execution: /usr/bin/ksh
4−>5: creating a child process

Figure 3: Fdformat attack.

D Ps attack - U2R

The typical form of ps attack involves both buffer overflow and race condition. The instances of ps attacks given in the MIT Lincoln Labs data set contain ps attacks that contain buffer overflow without race condition.

The `ps` attack takes advantage of a race condition in the version of 'ps' distributed with Solaris 2.5 and allows an attacker to execute arbitrary code with root privilege. This race condition can only be exploited in order to gain root access if the user has access to the temporary files.

The attacker needs to write an exploit program that deletes the `/tmp/ps/data` file. That action forces the ps program to create a new file and look in the `/tmp` directory for a file starting with ps. When it finds the file, it deletes and replaces it with a symbolic link to another file. The attacker will probably be forced to run the exploit many times before a success occurs. When the exploit is successful the ps will perform `chown` command on the symbolic link. The result is that the file the link points to is owned by root.

`Ps` attack included in the MIT Lincoln Labs data set is based on buffer overflow and has signature almost identical to previously described attacks. It can also be detected by looking at large values of header and path of execution which is `/usr/bin/ps` in this case.

The resulting diagram is represented on Figure 4.

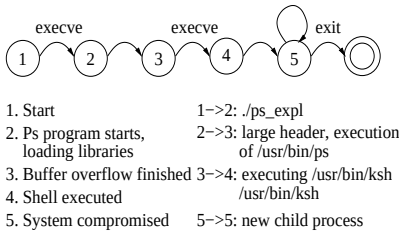


Figure 4: Ps attack.

III. Detection and classification algorithms with Hidden Markov Models

We need to recognize a malicious user who executes system calls in certain order in attempt to break into the computer system. In most cases of high-level attacks, sequences of system calls can be interleaved within normal system behavior in numerous ways.

Our goal is to detect when a particular sequence of harmful instructions that could compromise security of our system has been performed. To be able to recognize malicious activity we need to have a database of malicious actions and convert those actions in a set of HMMs, where each HMM is trained to recognize the attack pattern from which it was generated. Using the obtained information the IDS should perform matching of HMMs against possible attack sequences and find the likelihoods of the attacker's actions that were generated. The obtained likelihoods should be compared to obtain the information about the likelihood of attack or the likelihood that the given attack belongs to a specific class of attacks.

Before proceeding to our problem formulation and proposed solutions we present the overview of notation used for HMMs. The notation in this paper corresponds to the notation used in [3].

A discrete HMM is specified by the triple $\lambda = (A, B, \pi)$. A represents the state transition matrix of the underlying Markov chain and is defined as (where s_t denotes the state of the Markov chain at time t):

$$A = [a_{ij}] = [p(s_{t+1} = j \mid s_t = i)], i, j = 1, \ldots, N.$$

B is the observation matrix and is defined as (where x_t is the output (observation) at time t):

$$B = [b_{ij}] = [p(x_t = j \mid s_t = i)], i = 1, \ldots, N; j = 1, \ldots, K.$$

π is the initial state probability distribution of the underlying Markov states and is defined as

$$\pi = [\pi_i = p(s_1 = i)], i = 1, \ldots, N.$$

Given appropriate values of N, K, A, B and π the HMM can be used as a generator for an observation sequence. In our case we suppose that the number of states N and alphabet of observations K are finite.

To be able to use the algorithm that is presented next we need to compute the likelihood of a sequence of HMM observations given λ, $P(X \mid \lambda)$, which is done by using the forward variable. According to definitions in [3] the forward variable of an HMM is defined as

$$\alpha_t(i) = p(x_1, x_2, \ldots, x_t, s_t = i \mid \lambda) \tag{1}$$

Hence, $P(X \mid \lambda)$ is the sum of $\alpha_t(i)$'s. Recalling the solution of the forward part of the forward-backward procedure [3], $\alpha_t(i)$ can be computed recursively. The initial condition in the forward procedure is

$$\alpha_1(j) = \pi(j) b_j(x_1); 1 \leq i \leq N \tag{2}$$

and accordingly the recursive step [3] is

$$\alpha_{t+1}(j) = \left[\sum_{i=1}^{N} \alpha_t(i) a_{ij} \right] b_j(x_{t+1}); 1 \leq t \leq T-1; 1 \leq j \leq N \tag{3}$$

where $b_j(.)$ is the conditional density function given the underlying Markov state at time t is j.

We suppose that we have M HMMs as possible models for the observed data and we need to test which of the M HMMs matches the incoming sequence. In the framework of intrusion detection the problem can be formulated as follows: *given M attack models in the form of Hidden Markov Models with known parameters, detect the one that matches the incoming sequence with the highest probability.* However, in the case of detecting an attack the incoming sequence may or may not match one of the HMM models of attack. In case it does not match one of the attack models we need to consider two cases: either the incoming sequence is not an attack or it is an unknown attack that we don't have in our database. In this report we assume that the incoming HMM matches one of M HMMs in the system.

The problem of M-ary detection is solved with calculating log-likelihoods for each of the possible models given the observed sequence and finding the maximum. The model with the maximal log likelihood (closest to zero) wins and the attack is classified accordingly.

IV. Algorithm

Experiments are performed using the 1998 and 1999 DARPA/LL offline evaluation data. Both data sets had to be used due to the fact that there are no examples of `ps` attack in the 1998 data set. Lincoln Labs recorded program behavior data using the Basic Security Module (BSM) of Solaris. We used BSM audit records that correspond to system calls. Each record provides information like name of the system call, a subset of arguments, return value, process ID, user ID etc. We used the system call information and information about the payload size associated with each system call.

We developed an environment that would parse the BSM audit logs into a form that can later be used for detection and classification of attacks. The information about payload size and execution of an unexpected program was used for attack detection. Hypotheses testing was used for attack classification.

A Data processing phase

Step 1 The initial step deals with the whole BSM file that contains the daily activity performed on machine `pascal` and contains both normal and abnormal activity. We divide the whole BSM file into chunks of length of 100 system calls. Each system call is assigned a number according to the order of appearance in the BSM sequence.

Denote one randomly chosen system call number as k. The algorithm observes the payload size associated with each system call. If the system call has payload greater than 300, the program outputs $2*k$. Otherwise, it outputs $2*k-1$. The program lists the system calls with high payload that appeared in the BSM file. The algorithm also monitors for violations in the form of shell execution. If both conditions are fulfilled (oversized argument and execution of unexpected program) there is an ongoing attack.

Step 2 This step takes the original BSM file as an input and looks for instances of each attack. It produces meta-files containing line numbers of system calls issued by observed programs and the actual system calls with all parameters from the original BSM file.

Step 3 We extract sequences of system calls of length 100 and labels files that contain bad sequences. The resulting file contains system calls issued by the attacked program and some other system calls from other programs due to interleaving.

It may happen that the testing files (first and last) contain a large portion of normal sequences, which causes that the structure of training and testing file may significantly differ. That problem may be solved by a sliding window approach on training sequences.

B Training

In this paper we assume that the attack models are already known. The goal is to detect and classify the incoming sequence using the models generated on already known attacks. We trained the HMMs on attack sequences generated in the previous step. Each of the sequences is sequentially loaded and HMM parameters are generated. Each attack is represented with 4-6 sequences of length 100 but only one or two of them contain the actual attack (depends on whether the same sequence contains oversized argument and shell executions or not).

After sequentially loading the attack sequences and training parameters on them in 10 iterations we get log likelihoods for each model. After several iterations the parameters of trained HMM are found and A, B and π matrices don't change any more.

The training procedure consists of training each of malicious sequences on the same input parameters A_{in}, B_{in} and π_{in}, producing an output HMM with parameters π_i, A_i and B_i that characterize that specific malicious sequence (i denotes one of possible HMM models and varies in different training sets, but is usually less than 10). This property of HMMs that each set of parameters π_i, A_i and B_i fits a specific malicious sequence is used in the testing phase for classification of attacks in two levels: normal/abnormal and attack1/attack2.

C Detection

We concentrated on four programs: `ffbconfig, format, eject` and `ps`. The goal was to create hypotheses for each of the attacks and classify the attacks in the appropriate category. The classical rule for hypothesis testing when we set a threshold for each of the hypotheses could not be used. The hypothesis testing algorithm is based on the winning score rule, where winning score is the log-likelihood that is closest to zero. We denote the winning hypothesis with H_{WIN}. Hence, the hypothesis testing procedure is as follows:

$$H_{WIN} = \begin{cases} H_1 & \text{if } loglik_1 = \max_i\{loglik_i\}, i \in \{1,2,3,4\} \\ H_2 & \text{if } loglik_2 = \max_i\{loglik_i\}, i \in \{1,2,3,4\} \\ H_3 & \text{if } loglik_3 = \max_i\{loglik_i\}, i \in \{1,2,3,4\} \\ H_4 & \text{if } loglik_4 = \max_i\{loglik_i\}, i \in \{1,2,3,4\} \end{cases} \quad (4)$$

The first hypothesis, H_0 corresponds to normal behavior and that hypothesis is not used in the classification algorithm. The detection algorithm loads every sequence i, where $i = \{1, \ldots, N\}$ and N is the total number of sequences of length 100 in the given BSM sequence and tests whether the sequence has either shell execution or oversized argument. In either case, the sequence is classified as anomalous. The testing in this phase is performed between hypothesis H_0, that corresponds to normal sequence and H_A that corresponds to anomalous sequence. Hence, if the sequence contains either shell execution or has payload greater than 300, the winning hypothesis is H_A and the sequence is processed to the classification step. Otherwise, the winning hypothesis is H_0 and the sequence is declared to be normal and is not considered in further steps.

D Classification

This section tests on hypotheses H_i, $i = \{1,2,3,4\}$, that correspond to different types of attacks. In case of 4 buffer overflows the hypotheses are: H_1 (ffbconfig), H_2 (fdformat), H_3 (eject) and H_4 (ps).

Hypothesis testing was preformed on BSM chunks that were labelled as anomalous. The sequences were loaded into a matrix which was used for hypothesis testing. We created a number of HMMs fitted to `ffbconfig, format, eject` and `ps` testing sequences. Parameters obtained during training (π_i, A_i and B_i) were used as input parameters for testing. We calculated the log-likelihood for each of testing sequences using each of the models obtained by training as input parameters. If the tested sequence does not fit the malicious model, the log likelihood ratio converges to -∞ after the first iteration. Each sequence that had log likelihood greater than a certain (system-defined) threshold was classified as malicious. If a sequence is classified as both attack 1 and attack 2, two algorithms can be applied to avoid misclassification among the attacks.

Algorithm 1: This algorithm uses the property that the more similar two models are, the log-likelihood ratio is closer to zero. Hence, if a sequence is classified as both attack 1 and attack 2, the testing algorithm compares the log-likelihoods of the sequences and the one with larger log likelihood wins. This algorithm performed correct classification in the majority of cases.

Algorithm 2: This algorithm classifies sequences according to the program that is being executed in the sequence that was marked as anomalous. For example, is a certain sequence is classified as both attack 1 (attack on program 1) and attack 2 (attack on program 2) and our program determines that

program 1 is being executed in that sequence, the sequence is classified as attack 1. When we used this criterion there was no misclassification among the attacks. However, if we need to classify between a buffer overflow on `eject` program and some other type of attack on `eject` program this algorithm will not be of any use. The classification rate of this algorithm is 100%.

V. Results

All test performed on data sets include hypothesis testing. Only testing among `ffbconfig` and `eject` attacks and `ffbconfig` and `format` attacks is performed using 3 hypotheses: H_0 for normal, H_1 for `ffbconfig` and H_2 for `fdformat` and H_3 for `eject`. Week 6 Thursday contained 3 `ffbconfig` and 12 `eject` attacks. Week 5 Monday contained only one `ffbconfig` and one `fdformat` attack. Because of that, we performed training and testing on all possible combination of sequences and the results obtained were almost identical. Hypothesis testing using 5 hypotheses was not possible due to the fact that the testing data set does not contain any `fdformat` or `ps` attacks. For testing those attacks we trained each of those attacks on data from 2 days and tested it on data from the third day. For `ps` attack we used 1999 data.

A Detection of ffbconfig and eject attacks

Due to already presented difficulties, the hypothesis testing was performed using 3 hypotheses. Hypothesis H_0 corresponds to normal sequences, H_1 to `ffbconfig` attack and H_3 to `eject` attack. Testing was performed only on those sequences that had shell execution or even number (the sequences that were extracted in Step 1 of the algorithm). The results obtained in testing phase are first presented in the form of graphs and tables and then discussed.

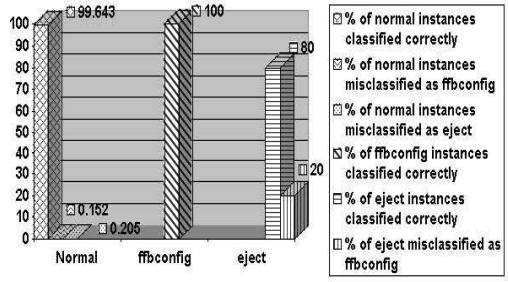


Fig. 5: Ffbconfig and eject tested on Week 6 Thursday.

	Normal	Anomalous
Normal	99.643%	0.357%
Anomalous	0%	100%

Tab. 1: Confusion matrix for the case of testing between 3 hypotheses: normal, ffbconfig and eject

Input parameters for tested sequences were HMM parameters obtained during the training phase. No eject attacks were classified as normal and no ffbconfig attacks were classified as normal. The only misclassification that happened was that some sequences were classified as both ffbconfig and eject attacks. There was a total of 9251 sequences. 9233 were normal,

	Normal	Ffbconfig	Eject
Normal	9200	14	19
Ffbconfig	0	6	0
Eject	0	3	9

Tab. 2: Detection and misclassification rates for the case of testing between normal, ffbconfig and eject

6 sequences characterized ffbconfig attack and 12 sequences characterized eject attack. When Algorithm 1 was applied to the sequences it led to the misclassification rate of 20% for eject attacks (8 out of 10 attacks were classified correctly) and 0% misclassification rate for ffbconfig. When Algorithm 2 was applied it led to misclassification rate of 0%. There were 0.368% of false positives and no false negatives. Hence, the only imperfection of the algorithm reflects in misclassification among different types of attacks, not among normal and anomalous. The results are presented in Figure 5 and Tables 1 and 2.

B Detection of ffbconfig and format attacks

Since there are no instances of format attack in weeks 6 and 7, we had to use two weeks from weeks 1-5 for training and one for testing. Applying the same procedure as in the previous case, the false alarm rate was 0.3% and misclassification rate was 0% using either of criterions presented in the previous section.

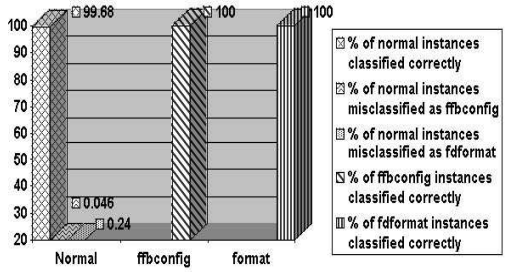


Fig. 6: Ffbconfig and format detection on Week 5 Monday.

False positive rate was 0.319% and there were no false negatives. No format or ffbconfig attacks were misclassified. The same results were obtained using both algorithms. The results are presented in Figure 6 and Tables 3 and 4.

	Normal	Anomalous
Normal	99.68%	0.3198%
Anomalous	0%	100%

Tab. 3: Confusion matrix in case of multiple hypothesis testing

C Detection and classification of ps attacks

Due to the fact that ps attack appears only in 1999 data, we could not test detection of ps attacks using any other hypotheses except H_0 (normal) and H_4 (ps). The false alarm rate was

	Normal	Ffbconfig	Format
Normal	99.68%	0.046%	0.24%
Ffbconfig	0	100%	0
Format	0	0	100%

Tab. 4: Detection and misclassification rates in case of detection and classification of multiple types of attacks

0.3% and there was no misclassification, all instances of ps attacks were detected.

	Normal	Ps
Normal	99.53%	0.47%
Ps	0%	100%

Tab. 5: Confusion matrix for detection of Ps attack

There were 0.47% false positives and no false negatives. The results are presented in Table 5.

The misclassification among the attacks is due to the facts that all buffer overflows have very similar structure and that chunking of BSM files does not always give us the whole attack, but instead includes parts of normal sequences. When we trained data for a specific attack and then used the data for detection of only that attack the detection rate was more than 99%. In table 6 we present tabular results in the form of confusion matrices for eject attack (trained on 3 weeks and tested on the fourth one), fdformat attack (trained on two weeks and tested on Week5 Monday) and ffbconfig attack (trained on 3 weeks and tested on Week5 Monday).

	Normal	Eject
Normal	99.577%	0.433%
Eject	0%	100%

	Normal	Fdformat
Normal	99.685%	0.0.315%
Fdformat	0%	100%

	Normal	Ffbconfig
Normal	99.577%	0.433%
Ffbconfig	0%	100%

Tab. 6: Confusion matrix for detection of Eject, Fdformat and Ffbconfig attacks respectively

VI. Conclusion

This paper demonstrated that it is possible to model network attacks with models that consist of small finite number of states. These models can be used for both detection and classification of attacks. Current results through our methods and algorithms do not display any False Negatives, but we cannot claim that False Negatives will not appear in future applications. The attacks for which we had most instances and associated data were attacks exploiting buffer overflows and that is the reason why our experimental results and applications of our algorithms presented in this paper are for such attacks. We developed models for other attacks and we believe that they are applicable for detection and classification of other attacks, not only buffer overflows. The algorithm can also be applied for detecting race condition attacks since they add additional loops in program behavior, that do not exist in normal behavior. The method presented in this paper can be used in combination with static analysis tools to achieve even higher detection rate and classify the attacks that the static analysis tools may have missed. The results obtained prove that the behavior of attacks can be completely captured by HMMs.

The contribution of this paper is reflected in the area of classification. There are numerous publications that use different models for detection between normal and anomalous. None of them puts emphasis on automatic classification of attacks. This paper uses an approach that is based on training on *anomalous* sequences. Training is performed on a couple of files, each of length 100 system calls. Testing is performed on around 2-4% of the total number of sequences of the initial data set. The strength of this approach is that it chooses only potential attacks in both the training and testing sets.

This method cannot be applied for detection of attacks that are performed over a long period of time. The strength of this approach is that the attacker cannot change the behavior of normal over time by slowly adding more and more abnormal sequences since we are using anomalous sequences for training.

This algorithm is used for detection of already known attacks. However, it can always classify the sequence as normal or anomalous. If the sequence is classified as anomalous and it does not fit any of the existing models, it can be named as *unknown* attack and passed on to another system that will determine if the attack is an already known attack that has been altered or a completely new attack.

References

[1] R. P. Lippmann et. al. "Analysis and Results of the 1999 DARPA Off-Line Intrusion Detection Evaluation", Recent Advances in Intrusion Detection 2000: 162-182

[2] B. Schneier, "Attacks Trees", Dr. Dobb's Journal, Dec. 1999.

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

[4] R. P. Lippmann, Robert K. Cunningham "Guide to Creating Stealthy Attacks for the 1999 DARPA Off-Line Intrusion Detection Evaluation", MIT Lincoln Laboratory. 24. April 2000.

[5] S. A. Hofmeyr, S. Forrest, A. Somayaji "Intrusion detection using sequences of system calls" Journal of Computer Security Vol. 6, pp. 151-180 (1998).

[6] C. Warrender, S. Forrest, B. Pearlmutter "Detecting Intrusions Using System Calls: Alternative Data Models", 1999 IEEE Symposium on Security and Privacy, May 9-12, 1999.

[7] R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni, "A Fast Automation-Based Method for Detecting Anomalous Program Behaviors", IEEE Symposium on Security and Privacy, Oakland, California,May 2001.

[8] D. Wagner, H. Chen "MOPS: an Infrastructure for Examining Security Properties of Software", 9th ACM Conference on computer and communications security, Washington DC, November 2002.

[9] D. Evans, D. Larochelle "Improving Security Using Extensible Lighweight Static Analysis", IEEE Software, January/February 2002.