Reducing false positives in intrusion detection by means of frequent episodes

Lars Olav Gigstad - 06HMISA

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2008

Avdeling for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Abstract

Intrusion detection system have become an increasingly important part of network security. Two types of intrusion detection systems are used, misuse and anomaly. Anomaly build a model of what is benign traffic, anything deviating from this will be flagged as malicious activity. Misuse search for pattern or known strings (a.k.a signatures) within network traffic, any matching traffic will be considered suspicious. How ever, often normal network traffic produce matches against signatures, creating large amounts of false alarms.

Data mining techniques looks for patterns or relations between records in a large data set. Frequent episodes is a data mining technique to find frequently occurring patterns, in an event sequence. In this thesis we apply frequent episodes as data mining technique to find patterns of frequently occurring alarms. From these patterns we create rules, which can be applied to filter out unwanted alarms. Experiment have been preformed on KDD Cup '99 data along with traffic from a small private network part of Gjøvik University College.

This thesis investigate the use of frequent episode data mining to find patterns in intrusion detection alert logs. Mining for frequent episodes is known for creating a lot of redundant and irrelevant episodes. We have applied our own algorithm to find and remove those episodes we do not want to keep. After we have removed unwanted episodes we are left with a small set of episodes from which we create attribute rules for.

Keywords: frequent episodes, frequent patterns, data mining, intrusion detection, false positives

Sammendrag

Innbruddsdeteksjonssystemer har i den senere tid blitt en viktig del av netverkssikkerhet. Disse systemene bruker enten signaturbasert eller anomliebaserte metoder for å oppdage angrep. Anomaliebaserte systemer bygger en model over hvordan normal trafikk er forventet på netverket. Trafikk som ikke passer inn i denne model blir antatt å være ondsinnet. Signaturbaserte systemer benyter seg av kjente mønstre som er representativt for angrepet. Normal netverkstrafikk kan ofte bli gjenkjent av signaturene som selv om de ikke er ondsinnet trafikk, dette gjør at innbruddsdeteksjonssystemer ofte rapporter en stor mengde med falske alarmer.

Datagraving er en teknikk som leter etter mønstre og sammenhenger mellom data i en stor data base. Frekvente episoder er en datagravings teknikk som finner mønstre som ofte opptrer i en sekvens med hendelser. Vi har brukt frekvente episoder til å finne alarm mønstre som ofte finner sted i en alarm log. Av disse mønstrene lager vi regler som blir brukt til å fjerne alarmer vi ikke er interresert i. I experimentene våre har vi brukt KDD Cup '99 sammen med traffic fra et lite netverk som er en del av Gjøvik Høyskole.

I denne masteroppgaven har vi brukt frekvente episoder til å finne mønstre i innbruddsdeteksjonssystemers alarm log. Denne teknikken er kjent for å finne mange frekvente episoder som ikke er relevante eller overflødige. For å løse dette har vi utviklet vår egen algorithme som finner og fjerner de episodene vi ikke er interresert i å beholde. Det vi sitter igjen med etter at uønskede episoder er fjernet er et lite set med episoder som blir brukt til å lage filtrerings relger fra.

Acknowledgements

I would like to thank my supervisor, Professor Slobodan Petrovic, for his help and guidance concerning the research topic. His input has been invaluable for the outcome of this thesis. My classmates at the masters lab (A220) at Gjøvik University College also deserve thanks for interesting discussions, and for making the work on this thesis enjoyable. Thanks to my fellow student Sverre Bakke too for the discussions and his valuable feedback regarding this report. I would also like to thank my friends and family for their support and encouragement during my work on this thesis.

— Lars Olav Gigstad, 16th of June 2008

Contents

Abstract . . . iii
Sammendrag . . . v
Acknowledgements . . . vii
Contents . . . ix
List of Tables . . . xi
List of Figures . . . xiii
List of Algorithms . . . xv
1 Introduction . . . **1**
1.1 Problem description . . . 1
1.2 Justification, motivation and benefits . . . 2
1.3 Choice of methods . . . 2
1.4 Terms and definitions . . . 2
1.5 Research questions . . . 3
2 Related work . . . **5**
2.1 Data mining . . . 5
2.1.1 Episode rules . . . 6
2.1.2 Association rules . . . 7
2.1.3 WINEPI . . . 8
2.2 Intrusion Detection . . . 9
2.2.1 SNORT . . . 11
2.2.2 BRO IDS . . . 11
2.2.3 Intrusion detection alerts . . . 12
2.2.4 The base-rate fallacy . . . 13
2.3 Data mining with IDS . . . 14
2.3.1 Spinning Cube . . . 14
2.3.2 ALAC - An Adaptive Learner for Alert Classification . . . 14
2.3.3 Root Cause Analysis to handle IDS alarms . . . 17
2.3.4 Custom filter using data mining . . . 18
2.3.5 MADAM ID . . . 19
2.3.6 Frequent episodes rules for internet anomaly detection . . . 20
3 Design and implementation of the system . . . **21**
3.1 Overview of our prototype . . . 21
3.2 Data preparation . . . 23
3.2.1 Intrusion detection alerts . . . 23
3.3 Data mining . . . 27
3.3.1 Frequent episodes . . . 27
3.3.2 Generation of candidate episodes . . . 29
3.3.3 Finding frequent episodes . . . 30
3.3.4 WINEPI Observation . . . 31
3.3.5 Removing unwanted episodes . . . 32

3.3.6 Attribute Rules 35
3.4 Analysis 37
3.5 Applying filter 37
3.6 Formal definition of algorithms 37
4 Data gathering 43
4.1 Experimental setup 43
4.1.1 KDD Cup ’99 44
4.1.2 Honeynet 46
4.1.3 Honeywall 47
4.2 Experiments 47
4.2.1 Experiment 1: Initial test 47
4.2.2 Experiment 2: True positive rules from honeynet 47
4.2.3 Experiment 3: False positive reduction 48
5 Results 49
5.1 Initial experiments 49
5.2 Experiment 2: Honeynet 50
5.3 Experiment 3: KDD Cup ’99 50
5.3.1 Week 4 51
5.3.2 Week 5 51
5.4 Discussion 55
6 Conclusion 59
7 Further work 61
Bibliography 63
A Source Code 67
A.1 Creating new serial candidates 67
A.2 Discovering frequent episodes 68
A.3 Remove unwanted episodes/Filter alert sequence 70
B Rules, Wednesday Week 4 73
C Installation 77
C.1 Bro IDS 77
D Installing HoneyWall 79

List of Tables

1 Window content as it is moved over where E and D are close 31
2 Window content as its moved over the sequence in Figure 17 32
3 Content of a window 34
4 Attributes of events A, B_1 and B_2. 35
5 Attributes of events A, B_1 and B_2. 35
6 Overview of installed software 47

List of Figures

1 Rough classification of data mining techniques [1] 6
2 Example episodes 7
3 Sequence of events [2] 7
4 Window in a sequence of events [2] 8
5 Snort overview, from www.snort.org 11
6 The Spinning Cube on a network with address space 128.55.0.0 - 128.55.255.255 15
7 Visual patterns found in the spinning cube 15
8 ALAC [3] 16
9 The genesis of root causes, or how root causes enter a system [4]. 17
10 Average alert reduction per IDS [4]. 18
11 Custom filter model [5] 18
12 Operation of MADAM 19
13 Detection rate from KDD Cup ’99 20
14 Prototype overview 21
15 Flow diagram of frequent episode algorithm 28
16 Sequence of events [2] 30
17 Sequence of events 32
18 Flow diagram of the episode removal algorithm 34
19 Filtered alert sequence 37
20 Episodes removed 49
21 Alerts in week 1 of KDD Cup ’99 52
22 Alerts in week 3 of KDD Cup ’99 53
23 Alerts in week 4 of KDD Cup ’99 54
24 Alerts in week 5 of KDD Cup ’99 56
25 Reduction at different thresholds 57

List of Algorithms

1 Rules algorithm 38
2 Main algorithm 38
3 Create new candidates 39
4 Finding frequent episodes 40
5 Remove unwanted episodes 41
6 Build rules 41
7 System algorithm 42

1 Introduction

The huge increase in the last 10 years in networked computers and Internet access in corporations, has led to an increased external and internal malicious activity. As one of the countermeasures, intrusion detection systems (IDSs) have become more popular the recent years. They try to detect attacks against a system violation of their security policy. With the deployment of IDS one of their weaknesses is becoming visible [4, 6, 7], false positives. False positives are alerts from an IDS on non malicious activity. With the increase in worm activity and more complex network structures the amount of alerts from IDSs have increased making it virtually impossible to check the validity of every alert [7].

There are two types of intrusion detection systems, anomaly and misuse detection. Anomaly learns what is considered normal traffic in a system and from this model alerts on any traffic deviating from this. Misuse detection on the other hand uses search for fixed patterns (i.e. signatures) within the traffic it knows is malicious. For both these systems a known problem is the high rate of false positives generated by these systems. False positives are alerts generated by the system on traffic that is not malicious.

To reduce these problems different data mining approaches have been used. Data mining are algorithm developed to search through a wast amount of data looking for patterns or relations, which can prove to be useful. These algorithms are often time consuming and require a lot of time to complete, but with the ever increasing processing power they are becoming more and more applicable. The patterns found are often surprising, and can provide further insight.

Frequent episodes is a data mining technique aimed at finding common patterns in a large sequence of events. An event can be described as a single action, input or system state. An episode is a collection of events occurring close together with a partial order. These episodes can then be used to describe or predict future events within the sequence.

1.1 Problem description

Properly deploying an intrusion detection system can be a time consuming task, and requires good knowledge of the architecture and environment it is deployed in. With highly dynamic environment in many networks today provided by wireless connection, having a properly configured intrusion detection system can be a difficult task. Network intrusion detection has been criticized for its ability to generate a large amount of false positives and false negatives[4]. The reasons for these large amounts of alerts is not caused by a single fault but a combination of several factors.

With a large amount of false positives the real threats detected by system can often go unnoticed as the system administrator will start ignoring reported incidents as the flood of alerts gets too high. This compromises the whole intrusion detection system reducing its security value to almost none.

1.2 Justification, motivation and benefits

With the increased use of intrusion detection systems as an integrated part of a security system, false alerts are becoming an increasingly larger problem. Responding to and evaluating an IDS alert is a labor intensive task, requiring vast human resource. With the increasing amount of alerts the task of handling these alerts are becoming a daunting task. Many system administrators often ignore or refuse to deploy intrusion detection systems due to their known high rate of alerts, missing out on many of the great benefits an intrusion detection system can provide in a network. We hope to develop with the help of data mining techniques algorithms which can find patterns in intrusion detection alert logs and create rules to filter away unwanted alerts, leaving a significantly reduced alert log.

1.3 Choice of methods

We have used a quantitative method based on both literature study and experimental work. In [8] they suggest starting with a thorough literature study to get a good background knowledge of what other researchers had done and their conclusions. With a review of earlier related work a good overview of problems, results and questions can be identified.

Theoretical research was a continuous process done in parallel with the development of our prototype. During this phase it was important to get a thorough understanding of how frequent episodes [1, 2, 9] worked and its problems. Since none of the algorithms for frequent episodes were available at the start of the project these had to be implemented before use. After implementation a thorough test phase had to be done ensuring the algorithms had been implemented properly, producing the wanted results. Further development of our prototype continued through our analysis phase, to help solving some of the problems found during our literature study.

Our experimental work consisted of using our developed algorithms to analyse alert logs. The results presented by our data mining algorithm were thoroughly verified before further analysis was done with the data presented. The results from data mining were used to help develop our algorithms further. The evaluation of the results was followed by a thorough investigation to verify the results presented by our system.

1.4 Terms and definitions

A more formal definition on terms used [2, 10, 11, 12, 13]:

- *Intrusion/Threat*,in a communication network is a potential event or series of events that could result in the violation of one or more security goals.
- *Intrusion detection*, is the process of monitoring the events occurring in a computer system or network, analyzing them for signs of security problems.
- *False positive*, alert raised on non-malicious activity.
- *True positive*, alert raised on malicious activity.
- *False negative*, no alert raised on malicious activity.
- *True negative*, no alert raised on non-malicious activity.
- *Attack*, is the actual implementation of a threat.

- *Sequence of events*, describing the behavior and actions of users or systems.
- *Episodes*, is a collection of events that occur relatively close to each other in a given partial order.
- *Honeypot*, is an information system resource whose value lies in unauthorized or illicit use of that resource.

1.5 Research questions

The research questions we will take a closer look at, are the following:

- Can alerts be effectively be correlated with frequent episodes?
- How effective is such an alert correlation?

2 Related work

2.1 Data mining

With the ever increasing processing and storage technology, more and more data are stored. Traditional analysis methods are not capable of handling these wast amounts of data. This is where data mining techniques help with, finding valuable information yet non-obvious. The need to find relations or patterns between data can be of great interest.

Data Mining is the analysis of (often large) observational datasets to find unsuspected relationships and to summarize the data in novel ways that are both understandable and useful to the data owner [1].

In the description above "observational dataset" refers to data being collected for other purpose than data mining. This is in contrast to statistics where experiments are often designed to find the answer to a specific question. Data mining techniques seek to reveal new useful information in a data collection. Figure 1 shows a rough outline of different data mining techniques.

Data mining has two major applications, decision support and program development. In decision support, data mining can reveal information in the form of rules or graphical representation. This helps to highlight common relations and trends within the large data set. A typical example is one found by grocery store data mining their purchases database for patterns; male customer shopping diapers on Thursday or Saturday, are more likely to buy beer. Used in program development, data mining can provide automated construction of activity or forecasting models.

Data mining is split into three basic phases:

1. Data preparation
2. Data mining
3. Analysis

Data preparation: Gathering the selected input and preparation of these values must be done before they are passed on to the mining algorithm. The quality of the mined results are affected by both the representation and quantity of raw data. How data is represented will affect the quality of the mined results. Input is usually mapped into 3 different categories:

- Scaled numeric values (e.g. range 0.0 to 1.0)
- Numeric values (e.i. integers)
- Categorical or Symbolic values (e.g. color, name)

Data preparation is used to give the mining algorithm data presented in an efficient way, reducing the overhead of this from the algorithm itself. Often data mining algorithms use data gathered from several sources. Data preparation gathers this into a single

data set. Mining algorithms often make several passes over the data set to find the final results. Having the data presented in a proper way helps reduce processing time.

Data mining: After the pre-processing, next phase is mining the data for results. There exist many data mining techniques and the selection of the appropriate algorithm is dependent on the data available and what we are interested in searching for. Data mining can be split into two main categories: *predictive* and *descriptive*. Predictive data mining techniques use training data to predict the value of one attribute based on the values of the other attributes. Predictive techniques find the relation between one attribute and the others. Neural networks and desicition trees are examples of predictive data mining. Descriptive data mining tries to find patterns and relations in a data set. It does not relay on training data opposed to predictive data mining. Patterns found are often described using rules based on statical significance from the data set.

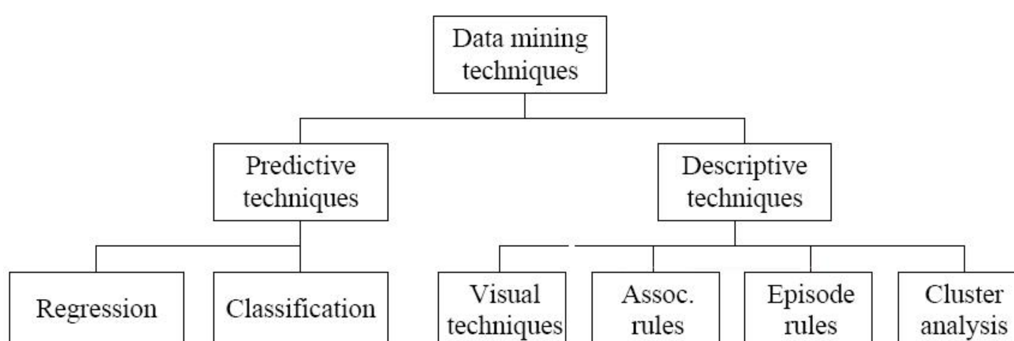


Figure 1: Rough classification of data mining techniques [1]

Predictive data mining is categorized into *regression* and *classification*. Regression uses attributes with numeric values to build predictions building a prediction of dependent variables[1] and their relation to one or more independent variables. Classification uses categories (i.e. color, name) where the order is not important, to build predictions.

Visual techniques use color and shapes to visually represent data. These are often graphs or maps representing the results. Clustering aims at putting data into groups, where the intra-distance between the elements in a cluster should be as small as possible while the inter-distance between clusters should be as large as possible.

Analysis: The final step is to analyze the results data mined. The result needs to be interpreted by a human analyst in order to find out how this new information can be used to improve decision making.

2.1.1 Episode rules

Episode rules describe the relations between events. An episode rule can be generalized into the following description:

> If a certain combination of events occurs within a time period, then another combination of events will occur within a time period [1].

[1] Dependent variables (y) are those that are observed to change in relation to independent variables (x), e.g. $y = f(x)$

More formally an episode $\alpha = (V, \leq)$, where V is a collection of events with a predicate $\leq$ as the partial order on V. Given a sequence S of events, α occurs in S if both the events in V and the partial order of $\leq$ are satisfied. Episodes can be represented by means of a directed acyclic graph [2], and can be divided into three main categories:

- **Serial**: Where a given set of events has a specific order in which the events occur.
- **Parallel**: There is no restriction on which order the events occur in.
- **Complex**: Episodes containing both serial and parallel events.

The Figure 2 shows three examples of different types of episodes. Episode α is a example of an serial episode. When A occurs it is followed by C shortly. Episode β shows a parallel episode. Which of A and B occurs first does not matter, when either has occurred the other will follow. The last episode γ shows a combination of both. When A and B have occurred in any order, C is likely to follow.

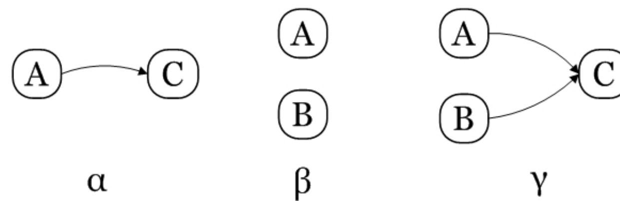


Figure 2: Example episodes

Figure 3 gives an example of what an event sequence might look like. For an episode to occur the events don't have to be directly following each other, there can be other event in between. They just have to appear in the correct order within the sequence to be classified as an occurrence of the episode.

Figure 3: Sequence of events [2]

An episode rule is an expression $\beta \Rightarrow \alpha$ where β is a subepisode of α ($\beta \subset \alpha$).

For an episode $\alpha = A \Rightarrow B \Rightarrow C$ to be frequent, each of the single episodes A, B and C have to be frequent. In addition, episodes $A \Rightarrow C$, $A \Rightarrow B$ and $B \Rightarrow C$ all have to be frequent. All these episodes are subepisodes of α. The subepisodes can be used to calculate the probability of a super episode occurring.

The rule observed above is used when creating new candidates to reduce the search space to as minimal as possible. The search is done in the reverse order, starting with the smallest possible episodes containing only one event and adding one more event until no more candidates can be made. When new candidates are made they are tested to see if they qualify as frequent according to the user preferences.

2.1.2 Association rules

While episode rules looked at the relation between events, association rules look at how two events or episodes occur together [2, 14, 15]. Association rules provide their rules

in an "if-then" manner, with a prerequisite (if) and a consequence (then). These rules are based on probability. To express the rules uncertainty, two values are used:

Confidence, the probability of the rule being true given it's prerequisite has occurred.

Frequency, how often the rule occurs in the data set.

Example:

$A \Rightarrow B$ with $conf(0.68), freq(0.28)$
This rule states that after A has occurred there is a 68% chance that B will follow. While this rule ($A \Rightarrow B$) occurs in 28% of the event sequences.

2.1.3 WINEPI

WINEP[1, 2, 9] is an algorithm developed by Mannila, Toivonen and Verkamo aimed at finding frequent episodes in a sequence of events. This method relies on moving a sliding window over the event sequence and finding how many windows each episode is present in.

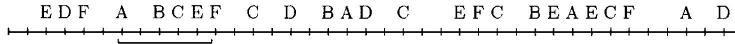


Figure 4: Window in a sequence of events [2]

Each event ϵ contains a set $A = a_1, a_2, ..., a_n$ of attributes and a time of occurrence t, $\epsilon = (A, t)$. A sequence of events $S = (s, T_s, T_e)$ is strictly ordered, where $e \in s$. The start and end time of the sequence are T_s and T_e, respectively. A window $\omega = (w, t_s, t_e)$ can be considered a sub sequence of S ($\omega \subset S$), where $t_s \leq T_e \cup t_e \leq T_s$, each event $\epsilon = (A, t) \in \omega$, where $t_s < t < t_e$, the size of a window is $|\omega| = (t_e - t_s)$.

Each episode can only occur once in each window. The frequency of an episode is defined as percentage of how many windows the episode occur in, or the probability the episode is in a randomly chosen window. The *frequency* of an episode α is formally defined as:

$$fr(\alpha, S, |\omega|) = \frac{|\{\omega \in W(S, |\omega|)|\alpha \in W\}|}{|W(S, |\omega|)|} \quad (2.1)$$

W is the collection of all windows in S. The confidence of the episode rule is given by Equation 2.2. Given episode β has occurred, this tells the probability that α will occur ($\beta \subset \alpha$).

$$conf(\beta, \alpha) = \frac{fr(\alpha, S, |\omega|)}{fr(\beta, S, |\omega|)} \quad (2.2)$$

The algorithm works by getting presented a set of candidates which are checked against the event sequence and those found to be frequent are kept. From these episodes new candidates are created. These candidates are then again checked to see whether they are frequent. This process is repeated until no new candidates can be created.

Lemma 1 If an episode α is frequent, then all sub episodes ($\beta \subset \alpha$) are at least equally frequent.

The generation of candidates is done with a breadth-first search following the relation between sub episodes and episodes given in Lemma 1 above. The search starts with most basic episodes, with only one event and grows the episode larger for each round until no more candidates can be created.

Example:

> $A \Rightarrow B$ with $win(60)$ $conf(0.68), freq(0.28)$
> This rule states that after A has occurred there is a 68% chance that B will follow within 60 seconds. This episode occurs in 28% of the event sequence. A and B can be both single events or episodes.

2.2 Intrusion Detection

Intrusion detection systems are used to monitor network traffic and generate alerts based on potential malicious traffic. There are two main categories of intrusion detection systems: signature and anomaly based. Signature based systems inspect every packet and look for a specific pattern, very much like how virus scanners work. Anomaly based systems try to learn what traffic is normal. Anything deviating from this is considered harmful. One of the drawbacks with anomaly based intrusion detection is that it has a hard time categorizing traffic it finds malicious. Intrusions can be divided into 6 main types[16]:

- Attempted break-ins, which are detected by atypical behavior profiles or violations of security constraints.
- Masquerade attacks, which are detected by atypical behavior profiles or violations of security constraints.
- Penetration of the security control system, which are detected by monitoring for specific patterns of activity.
- Leakage, which is detected by atypical use of system resources.
- Denial of service, which is detected by atypical use of system resources.
- Malicious use, which is detected by atypical behavior profiles, violations of security constraints, or use of special privileges.

The intrusion detection can be split into different categories based on: detection technique, scope and action taken. By looking at detection technique, intrusion detection systems can be categorized into two categories according ti the techniques used. In many systems these are not mutually exclusive, meaning that many IDSs contain both detection methods.

Anomaly Detection: These system work by assuming benign and intrusive traffic has different characteristics. By building a profile of what is considered normal traffic, we can flag all traffic varying from this profile by a statistically significant amount as an intrusion. Anomaly detection systems are also computationally expensive, to keep track of and update several attributes. Due to the way these systems detect attacks they rarely provide very detailed information about the attack. One of

the main advantages of anomaly detection is the possibility of detecting currently unknown attacks, if they deviate from the normal traffic profile.

Misuse Detection: The concept behind misuse detection is that attacks can be represented by a pattern or common sequences within the traffic. These are often called signatures. Misuse detection systems can only detect attacks they have signatures for. For unknown attacks they provide little help. For found attacks a misuse based system is able to provide detailed reports about the alarm and what caused it, which is an advantage compared to anomaly detection. A big problem for misuse based systems is writing correct signatures, which should include all possible variants of an attack but at the same time they should not match non-intrusive activity. This is explained further in Section 2.2.3.

Scope of protection is what the intrusion detection system is trying to protect. There are many different categories in scope of protection, we here present only the most common [10]:

Network Intrusion Detection: These intrusion detection system are directly connected to the network inspecting all passing traffic. These system often inspect the packets at a low level in OSI model, the transport and network layer. A single system is able to monitor traffic to several host on a network.

Host Intrusion Detection: Host-based intrusion detection system are often installed on a single host it is trying to protect. Inspecting audit and system logs from the operating system and other information available.

Protocol Intrusion Detection: Protocol-based intrusion detection system are often installed on hosts using applications protocols (e.g. web servers). These systems have a deeper knowledge of the protocol and can often analyse encrypted traffic if the protocol supports it (e.g. HTTPS). PIDS target a single set of protocols trying to detect any malicious activity on any of these.

How an intrusion detection systems react to an intrusion can be categorised into two different categories:

Passive: These are what most consider when talking about intrusion detection systems. These systems will only write a reports of each incident they detects. A system analyst can then later investigate these reports and check what has occurred. A disadvantage of these systems is that the attack has often been completed before the analyst has had time to check and respond to the incident, unless the alert log is monitored 24 hours every day.

Reactive: Often referred to as intrusion prevention system (IPS). IPS can be configured to automatically reconfigure security equipment to block malicious activity it detects. They stop attacks before they have the chance to complete. The drawback of this system is high false positives from benign traffic may be blocked reducing the productivity of the network.

2.2.1 SNORT

SNORT is an open source IDS based on signatures. SNORT is focused on performance, simplicity and flexibility. It is based on the portable libpcap packet sniffing library. All the initiation and rule parsing is done before packet capture starts to make sure the overhead for each packet processing is kept at minimum. Snort consist of four main components:

- Packet decoder
- Preprocessor
- Detection engine
- Logging and alerting subsystem

SNORT can be configured to run in four modes [17]:

- Packet Sniffer mode, which simply reads the packets off the network and displays them in a continuous stream on the console (screen).
- Packet Logger mode, which logs the packets to the disk.
- Network Intrusion Detection System (NIDS) mode, the most complex configuration, which allows SNORT to analyze network traffic for matches against a user-defined rule set and performs several actions based upon what it sees.
- Inline mode, which obtains packets from iptables instead of from libpcap and then causes iptables to drop or pass packets based on Snort rules that use inline-specific rule types, making it work more like a IPS[2].

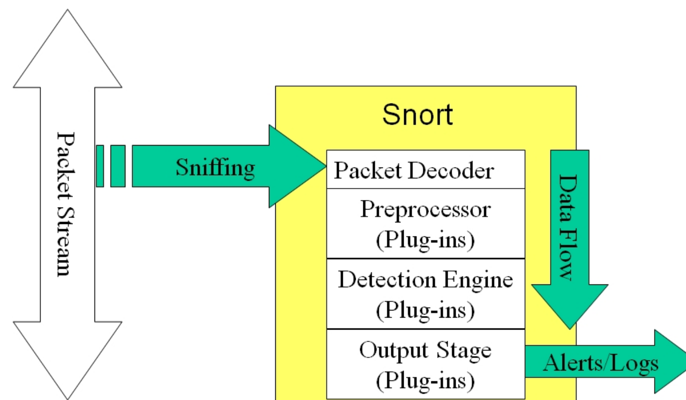


Figure 5: Snort overview, from www.snort.org

2.2.2 BRO IDS

BRO [18] is another open source Network IDS for Unix-based systems. Bro was developed by Vern Paxson of Lawrence Berkeley National Labs and the International Computer

[2]IPS: Intrusion Prevention System

Science Institute. It has been primarily developed as a research platform for intrusion detection and traffic analysis. BRO is divided into three layers:

- *libpcap*, packet capture.
- *event engine*, reduces the packet stream into a series of events.
- *event handlers*, policy scripts.

BRO interprets the structure of the network packets and converts them into events describing the activity. This stream of events is then run through the policy scripts. Policy scripts are programs written in the BRO's own scripting language. They are used to analyze the network events and take the appropriate actions. Events are actions that take place on the network. Examples of events might be a failed connection attempt, a connection established between two computers or a successful authentication.

It also contains a signature matching capability that performs content checking of the packets. BRO uses regular expressions in contrast to SNORT's fixed string matching, in signatures. BRO is also compatible with SNORT's signatures and comes with a tool to convert SNORT signatures to BRO's format. One of the interesting things with BRO is all the log files it creates from traffic. BRO has several different logs where some are more of the common type by containing alerts and signature matches, while others contain worm activity and connection records.

2.2.3 Intrusion detection alerts

Intrusion detection systems can generate hundreds of alerts per minute depending on different factors. The amount of alerts an IDS generates is a serious and complex issue. While we would like the IDS to alert on traffic that is malicious, creating too many alerts can have the opposite effect. To many false positives make it hard to spot the real threats. Creating a flood of alerts an administrator has to handle, he might become overwhelmed and start skipping or ignoring alerts altogether, making the IDS security value worthless. Some common causes of alert flood are:

- Some signatures poorly describe the attack making them trigger on benign traffic as a result. Instead of using regular expressions to describe an attack, simple string matching is often used. This has several reasons; due to processing time restrictions simpler string-matching is used to be able to keep up with traffic data [19]. Writing good and correct signatures is a difficult task and often leads to buggy or incomplete signatures [20].
- Some signatures trigger on rare or suspicious traffic. These in themselves are not classified as attacks but are considered uncommon, i.e failed logins, overlapping IP fragments or the use of URGENT bit in the TCP header. It has been shown that these alerts often are not linked to malicious activities [18].
- Alerts trigger on low-level phenomena, scanning every single packet and reporting on every packet the system finds malicious. Often attacks consist of several steps to complete, resulting in a single attack consisting of several alerts representing a single attack. This is mostly due to the lack of overview an IDS has [3, 4].

- The architecture of the network and it is layout can trigger alerts. A network with a low MTU[3] will often fragment packets. Most IDS will alert on every fragmented packet creating a flood of alerts [4, 21].

2.2.4 The base-rate fallacy

To give a more formal description of the false positive problem, we can use the base-rate fallacy. The base rate fallacy is an important part of Bayesian statistics, from Bayes' theorem stating the relationship between two probable events.

$$P(A|B) = \frac{P(A) \cdot P(B|A)}{P(B)} \tag{2.3}$$

Base-rate is easiest described with an example. Let's say we have a test which has an accuracy of 99.9% of being correct. This might seem like a good test with such a high accuray. Only one of every thousend will be wrong. The problem starts when the population gets very large, and only a fraction of the population should actualy test positive to the test. Taking an intrusion detection system as an example, it is not unusal to audit several billions of records (e.g. packets). Less then a thousaud of these might be actual maliciouse activity (i.e. gives a probability of 1/10,000,000 for maliciouse activity). By continuing our example, let A or $\neg A$, be the presence of an alert or not, while I and $\neg I$ denotes an actual intrusion or not.

	Positive	Negative
True	$P(A\|I)$	$P(\neg A\|\neg I)$
False	$P(A\|\neg I)$	$P(\neg A\|I)$

Our main interrest lies in:

- $P(I|A)$, those alerts that correctly indicate an intrusion.
- $P(\neg I|\neg A)$, the absence of alerts correctly indicates no intrusion.

When we apply Bayers' theorm to the two probabilities above we get the following equations[6]:

$$P(I|A) = \frac{P(I) \cdot P(A|I)}{P(I) \cdot P(A|I) + P(\neg I) \cdot P(A|\neg I)} \tag{2.4}$$

$$P(\neg I|\neg A) = \frac{P(\neg I) \cdot P(\neg A|\neg I)}{P(\neg I) \cdot P(\neg A|\neg I) + P(I) \cdot P(\neg A|I)} \tag{2.5}$$

Now let a intrusion detection system audit 1,000,000 records (e.g. network packets), which contains 2 intrusion each consisting of 5 records.

$$P(I) = \frac{1}{\frac{1 \cdot 10^6}{2 \cdot 5}} = 1 * 10^{-5} \tag{2.6}$$

$$P(\neg I) = 1 - P(I) = 0.99999 \tag{2.7}$$

By inserting these values into the Equation 2.6 we get:

[3] MTU: Maximum transmission unit, the size of the largest packet a network protocol can transmit

$$P(I|A) = \frac{1 \cdot 10^{-5} \cdot P(A|I)}{1 \cdot 10^{-5} \cdot P(A|I) + 0.99999 \cdot P(A|\neg I)} \quad (2.8)$$

By studying the above equation we see the outcome is dominated by $0.99999 \cdot P(A|\neg I)$. While the best possible outcome is with $P(A|I) = 1$ and $P(A|\neg I) = 0$, these values are practically impossible to achieve. With more normal detection rate of $P(A|I) = 0.8$ but with a still low false positive rate of $P(A|\neg I) = 1 \cdot 10^{-5}$ we get $P(I|A) = 0.50$ resulting in half the alerts generated being false positives. Now given our example only used 1,000,000 inspected records, which can be considered low in todays environment where traffic can generate a magnitude of hundred or even thousands times more audit records. The problem only gets worse.

2.3 Data mining with IDS

Using data mining with intrusion detection system isn't something new. KDD-Cup competition [22, 23, 24], challenged participants to learn to classify connections into "normal" and "intrusion", based on a training set containing all attacks. The most apparent drawback of this, is it's unlikely to find a practical training set which can replicate a production network and to include all possible attacks. Below we'll give a short summary of other research experiments done with data mining with intrusion detection.

2.3.1 Spinning Cube

In [25] an experiment with visual data mining on Bro's logs was performed. Using the log files created by Bro containing information about every connection observed, attempted and successful. From this a cube is created where the X-axis represents our network address space (e.g. 128.39.44.0/24), i.e. destination address. Z-axis represents the whole address space (0.0.0.0 - 223.255.255.255), i.e. addresses of incoming connection, multicast traffic (224.0.0.0-255.255.255.255) are not displayed. The Y-axis represented the port number (0-65535).

Extracting this information creates a single point for each connection. Successful connections (SYN/FIN) are shown as white dots while multi-colored dots represent incomplete TCP connections, SYN/RST or SYN with no response. The incomplete connections are colored using a colormap with the color varying by port number. This helps the viewer locating the point in a 3d space. The cube is represented in 3d space and it is possible to rotate the cube to get different perspectives.

The most interesting finding from this experiment are some of the patterns emerging from the traffic. One of the patterns, which they called "Barber poles" is shown in Figure 7(a). These scans vary their port number and IP addresses in an attempt to evade detection. Another type of scan that was detected was dubbed a "Lawnmower" shown in Figure 7(b). These scans are quite "noisy" in that they scan a wide range of contiguous ports across the address space.

2.3.2 ALAC - An Adaptive Learner for Alert Classification

Adaptive Learner for Alert Classification[3], is a system for reducing false positives in intrusion detection. The system classifies every alert as either *true positive* or *false positive*, along with a *classification confidence* value. The paper bases its work around two assumptions:

Analyst's knowledge is implicit: Analysts find it hard to generalize, i.e., to formulate

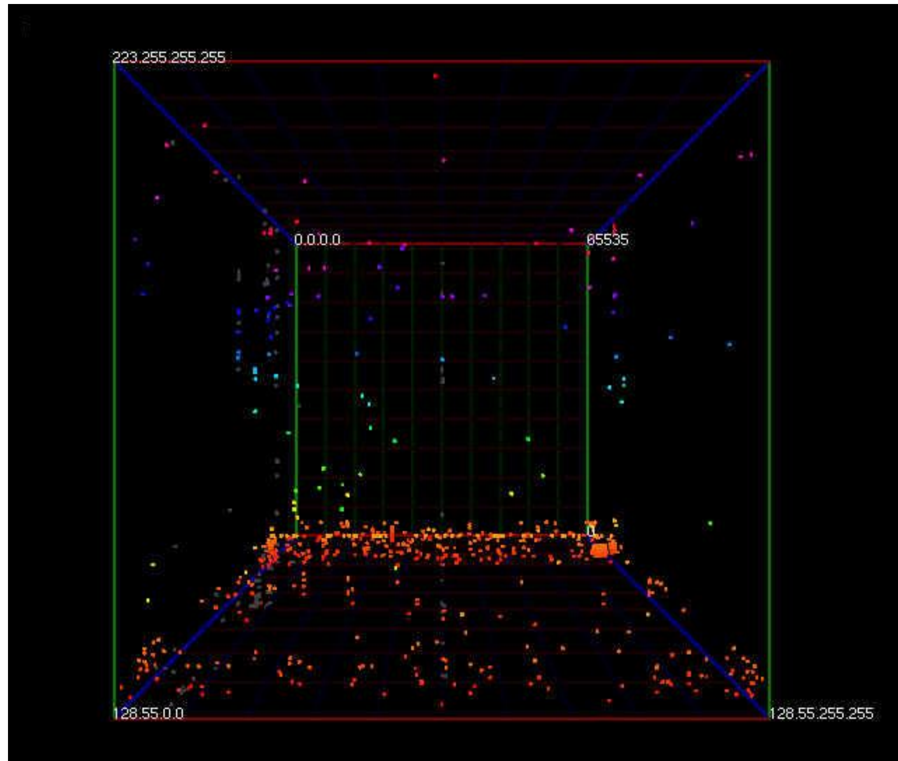
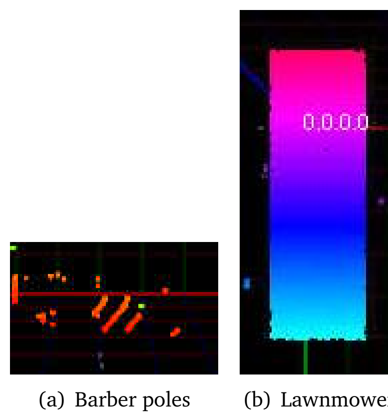


Figure 6: The Spinning Cube on a network with address space 128.55.0.0 - 128.55.255.255



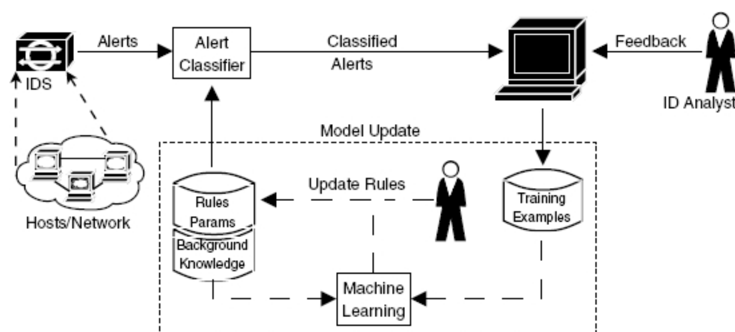
(a) Barber poles (b) Lawnmower

Figure 7: Visual patterns found in the spinning cube

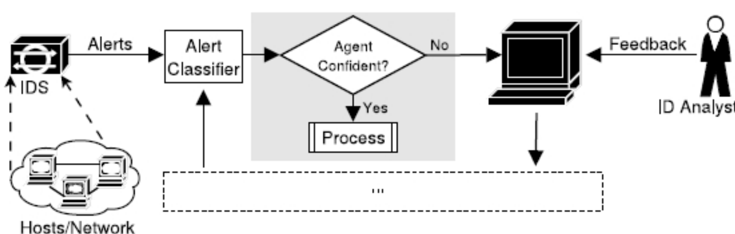
more general rules, based on individual alert classifications. For example, an analyst might be able to individually classify some alerts as false positives, but may not be able to write a general rule that characterizes the whole set of these alerts.

Environments are dynamic: In real-world environments the characteristics of alerts change, e.g., different alerts occur as new computers and services are installed or as certain worms or attacks gain and lose popularity. The classification of alerts may also change. As a result, rules need to be maintained and managed. This process is labor-intensive and error-prone

The knowledge of how to classify alerts is learned by the feedback from the analyst. ALAC can be set up to work in two modes: recommender (a) and agent mode (b). Recommender mode works by passing all alerts along with a classification. Confidence tells the probability of the classification to be correct. These are presented to the analyst that can reclassify if desired. This feedback is recorded and used to further improve the alert classifier. As a rule learner algorithm RIPPER was chosen. ALAC classifies alerts in real-time and updates it's learning data. In agent mode alerts with high confidence, user-defined actions can be performed. Alerts with a high true positives could be used to reconfigure a firewall or router, while false positives could be ignored. Alerts with a low confidence value would still be pass along to an analyst for feedback and help improve further classification.



(a) Recommender mode



(b) Agent mode

Figure 8: ALAC [3]

They used two data set to test their KDD Cup '99, and one they collected from a private network not connected to the internet. Results showed background knowlage had a larger impact on the KDD Cup dataset then from their real world experiments. Their experiment reduced false positives with approximatly 30%

2.3.3 Root Cause Analysis to handle IDS alarms

This thesis looks at the causes of high rates of false positives. Every alert occurs for a reason, which is referred to as the alerts root cause.

Definition A root cause is a problem that affects one or more components and causes them to trigger alarms. Root cause analysis is the task of identifying root causes as well as the components affected by them.

They show a quick overview of different root causes and how they can affect a system. This overview is in no way a complete overview or an attempt at classifying the the root causes.

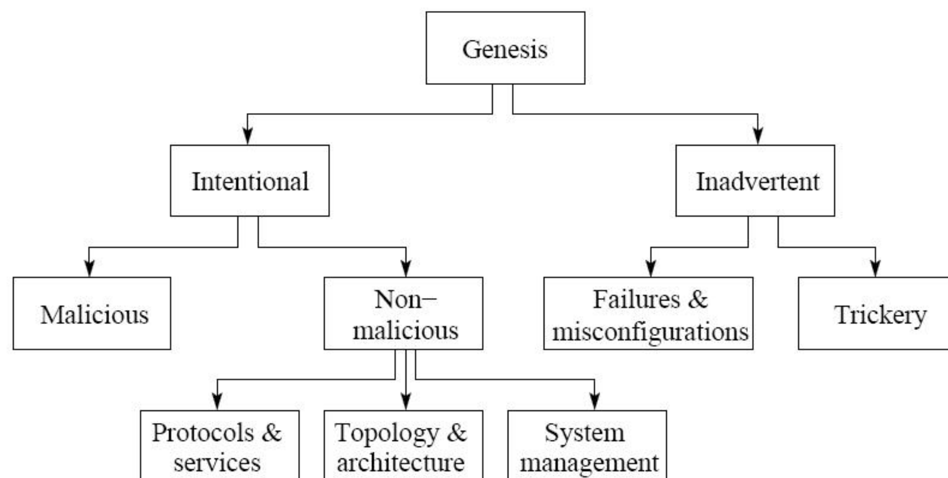


Figure 9: The genesis of root causes, or how root causes enter a system [4].

Klaus Julish [4] observed that over 90% of alerts in an alert log are cause by a few dozen root causes. Generally these are persistent, and keep triggering alerts until they are removed. Further it is shown that alerts triggered by the same root cause often share some structural properties. Based on these obervations, they propose alert clustering as a way of grouping alerts together. After these cluster have been found they are presented to a human analyst which would be reponsible for finding the cause. When the root cause is identified, the alerts can be filtered out. The experiment from [4] shows that finding and removing root causes can reduce the alarm load with 70% on average. The work in this thesis proved the three following statments[4]:

1. A small number of root causes is generally responsible for a lot of alerts triggered by an IDS.
2. Root causes can be discovered efficiently by performing data mining on alert logs.
3. Knowing the root causes of alerts, one can in most cases safely and significantly reduce the future alarm load by removing them or by filtering out the alerts that originate from benign root causes.

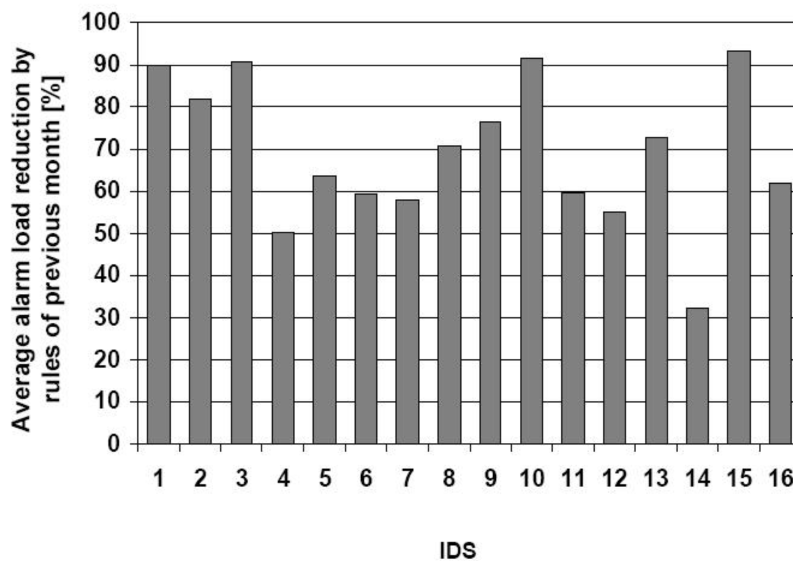


Figure 10: Average alert reduction per IDS [4].

2.3.4 Custom filter using data mining

Chris Clifton and Gary Gengo [5] used data mining to find episode rules within an alert log and create custom filter rules. These can filter well understood false positives. They analysed the alert log for event sequences of size 2-5, with the same destination and source address.

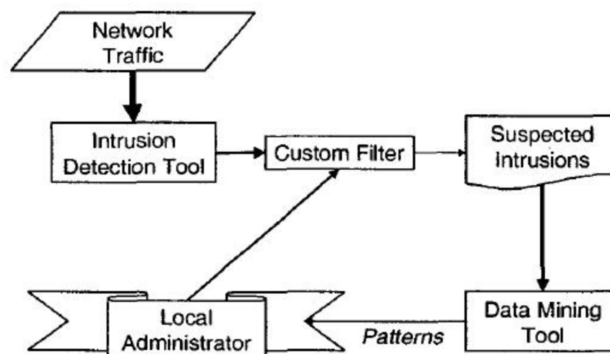


Figure 11: Custom filter model [5]

They show only very limited preliminary results in Figure 11. Most of their results show it is possible to find correlation between alerts. Their is no discussion on how effective it is or how they filter their the alert log with the rules found. The research is done in corporation with MITRE and the U.S. Army's Communications Electronics Command and their report might have been restricted by confidentiality.

2.3.5 MADAM ID

Mining Audit Data for Automated Models for Intrusion Detection [26, 27] was one of the first projects to use data mining with intrusion detection. MADAM ID builds an IDS from data mining data traffic on a network. It does not consider the payload of the packet, instead all traffic is abstracted to connections records. These records contain attributes like: Source IP, Destination IP, Port numbers, start time, duration, header flags, etc.

MADAM relies on training data. This data set has to be split into *normal connection records* and *intrusion detection records*. On the training data MADAM preforms two steps: *feature-construction* and *classifier-learning*. During the classifier step a misuse detection IDS is built while the first step builds an anomaly detection system. Figure 12 give an overview of the process.

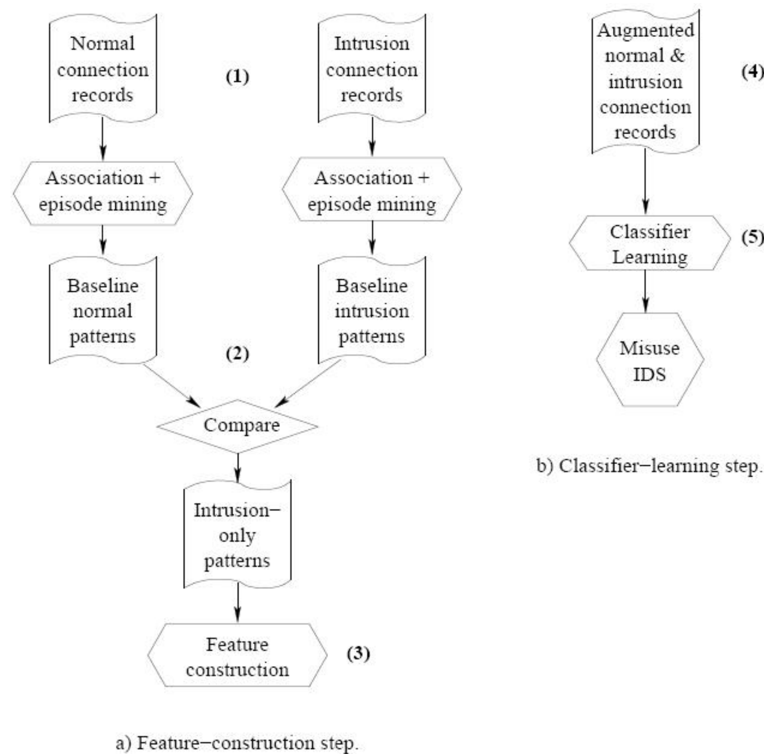


Figure 12: Operation of MADAM

1. The training data for MADAM ID must be split into two parts, *normal connection records* and *intrusion detection records*. MADAM ID offers no support for this.
2. Association rules and episode rules are mined from the normal connection records and from the intrusion connection records. The resulting patterns are compared, those unique for the intrusion connection records are used to form the intrusion-only patterns.
3. Intrusion-only patterns are used to find additional attributes, which are expected to be indicative of intrusive behavior. These additional attributes are counts, averages, and percentages over connection records that share some attribute values with the current connection record.

4. The original training connection records are augmented by the newly constructed attributes.
5. A classifier is learned that distinguishes normal connection records from intrusion connection records. This classifierthe misuse IDS is the end product of MADAM ID.

MADAM ID relies heavily on expert knowledge in intrusion detection. This knowledge is used to prune the number of patterns produced during association and episode rule mining.

2.3.6 Frequent episodes rules for internet anomaly detection

Min Qin and Kai Hwang [28] used frequent episodes (*WINEPI*) to create rules from TCP, UDP and ICMP traffic, these rules form an anomaly detection system. They used KDD Cup '99 data set along with traffic captured from University of Southern California. From KDD Cup '99 they used week 1, 3, 4 and 5. To form an event they used srchost, desthost, srcport and service as essential attributes identifying each connection.

They applied a few ways to remove unwanted episodes, these where based on observations of how their algorithm generated the episodes, and their relevance. They focus on making the episodes as small as possible, reducing the rules made by total of 40% to 70% depending on the window size used. By keeping the episodes as small as possible they argue that several of the larger episodes can be modeled by combining the smaller ones. Their method for removing rules did not affect the false negatives rate of their system. With KDD Cup '99 they achived an average detection rate of 31%.

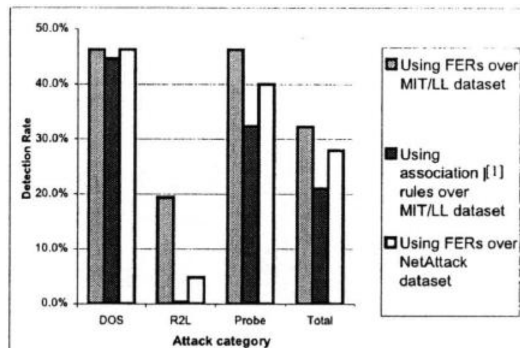


Figure 13: Detection rate from KDD Cup '99

3 Design and implementation of the system

In this chapter we describe our approach to finding false positives in intrusion detection logs. None of the algorithms were available prior to the thesis and had to be implemented and tested before use. All of the algorithms have been implemented in C++. To find patterns we used *WINEPI*, which mines both episode rules and association rules. The algorithms used will be described in detail. Our choice to use frequent episodes is based on the following observations:

1. False positives from normal traffic are expected to trigger the same alerts several times.
2. By analyzing frequently occurring alerts the analyst will have large impact on the alert log with minimal time used.

We limited ourselves to the use of *WINEPI* since proper implementation and testing of the algorithm is time consuming. With *MINEPI*[9] building on the same framework of algorithm as *WINEPI* getting a proper implementation of this first was a priority. *WINEPI* presented in Section 2.1.3 has the ability to find patterns in the form of episodes and association rules between the episodes by using a sliding window. To not limit ourselves to only find episode based rules, but also to find a more strict correlation, we applied additional mining to the attributes of the alerts within an episode. Stricter rules makes it less likely to discard true positives [4].

3.1 Overview of our prototype

Figure 14 shows an overview of how our system works. This Chapter will mainly focus on the algorithms used in the data mining part and the different phases involved in creating the final rules. These rules are presented to a human analyst before the finally decision about applying the rule or not should be done. Accepted rules will be applied in a filter removing alerts from the alert log matching the criteria of the rule.

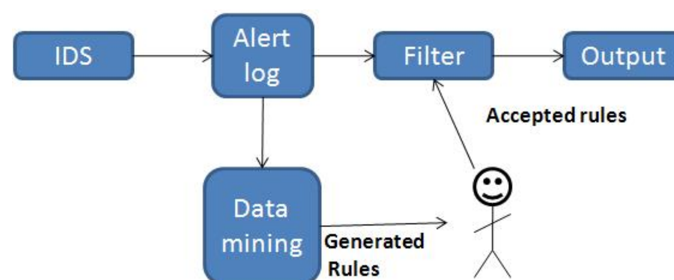


Figure 14: Prototype overview

As opposed to other models [3, 5] where the rules act as feedback by actually discarding alerts before they are written to the alert log, we opted to use a model where the filter is applied after the alert log acting more as a display filter. The main reason

for applying the filter after the alert log, is because of the time window our algorithm uses. With our filter applied before the alerts are written a log would create a time delay equal to the size of our window. Further the observations described in Section 3.3.4 and our adaptation of using several mining steps to complete our rule generation, removing alerts from the alert log would alter the way our algorithm generated rules making it less likely to adopt to changes in the network architecture. Using it as a display filter we can still display alert within the window, and the probable rule matches. Leaving the alert log with all its original data opens up for other types of analysis.

As described in Section 2.1 data mining consists of three phases. We will follow these phases and explain what is done in each and how the algorithms we have applied work in more detail. Below is a quick overview of what happens in our data mining algorithm, and what each phase includes.

From the initial research a common problem with mining for episode rules is the large amount of irrelevant or redundant patterns found [29, 30, 28]. Step 2 was added to help minimize this problem and remove those episodes we did not want. For this we developed our own algorithm based on our knowledge and observations from using episode mining.

Step 3 adds stricter properties to the rules. As Julish [29] discusses the more specific a rule is the less likely it is to discard true positives. Restricting rules based on the properties of an alert makes them more specific to a single cause of an alert.

- Data preparation
  - Parse alert log, (SNORT or BRO)
- Data Mining
  - Step 1, Find frequent episodes
  - Step 2, Remove unwanted episodes
  - Step 3, Build attribute rules
- Analysis
  - Present rules

Quick overview of notation used in examples given during the course of this Chapter, and the algorithms in Section 3.6.

A: event (i.e. alert)

α: episode

β: sub episode of α ($\beta \subset \alpha$)

C: Collection of candidate episodes [1]

F: Collection of frequent episodes [2]

C_l or F_l: Collection of episodes with size l, $l = |\alpha|$ where $\alpha \in F_l$ or C_l.

[1] Considered an array, with the first episode at $C[1]$.
[2] Considered an array, with the first episode at $F[1]$.

W: Collection of alerts within a window.

$|\alpha|$: Size of an episode, number of events it contains [3]

3.2 Data preparation

In this phase we extract the data we need from each alert, and present the data in a way our data maining algorithm can further analyse the data. We will start by looking at what information SNORT and BRO alerts contain. We then continue with how this information is used, and converted to an event sequence.

3.2.1 Intrusion detection alerts

When it comes to IDS alerts, there is no standard on the format or what they should contain. Most IDS logs contain a few common attributes though:

- *id*, unique identifier for the type of alert.
- *source*, IP address of the host sending the packet.
- *source port*, from what port on the source it was sent from.
- *destination*, IP address of the victim.
- *service*, destination port on the victim.
- *time*, time of occurrence.

We have chosen SNORT as our main IDS. But we have also looked at BRO-IDS. SNORT can log alerts to wide range of formats, from databases to text-files. For this experiment we were logging our alerts to text files as neither the traffic nor the amount of alerts is expected to be very high. Next we'll take a closer look at a SNORT alert and look at what information they contain.

```
[**] [1:1384:10] MISC UPnP malformed advertisement [**]
[Classification: Misc Attack] [Priority: 2]
04/12-16:52:40.841230 128.39.243.170:2250 -> 239.255.255.250:1900
UDP TTL:4 TOS:0x0 ID:65104 IpLen:20 DgmLen:242
Len: 214
[Xref => http://www.microsoft.com/technet/security/bulletin/MS01-059.mspx]
```

Each alert can consist of several lines, containing attributes about the alert. The "[**]" at the start and end of the first line does not serve for any important purpose other than making it easy to spot the start of new alerts. The next field is an ID that consists of three numbers: [*GID*:*SID*:*RID*].

- *GID:* Generator ID, tells what component of SNORT generated the alert.
- *SID:* Signature ID, the SID number is written directly into the rule by using the SID option within SNORT signatures.
- *RID:* Revision ID, tells what revision of the signature was used.

[3] Considered an array, with the first event at $\alpha[1]$.

The next word will be in uppercase, and tells which rule file it comes from. In this case the signature for the alert can be found in "misc.rules". The rest of the text string on the first line, gives a quick note of what the alert was about. The second line tells what classification the alert belongs to and what priority it has. Snort uses four levels of priority:

1. High
2. Medium
3. Low
4. Information

Priority is often based on which classification it comes from, but can be overridden for each signature. It is a preset value determined by the signature writer and is therefore a subjective value. The third line gives a date and time stamp of the time of occurrence. Next follow source and destination addresses. The fourth line contains a lot of detailed info about the packet and its headers. First column gives the protocol used. It will in most cases be TCP or UDP. Next is the TTL[4] field from the IP header in the packet, TOS[5] also from the ip header. *IpLen* and *DgmLen*[6], tells the size of ip header and the data sent respectively. The last lines contain URL address where the vulnerability is described in more detail.

From the attributes above we have chosen the following ones to be used in our system:

- *alert id*, the type of alert.
- *source*, IP address of the attacker.
- *destination*, IP address of the victim.
- *source port*, source port on the attacker.
- *destination port*, destination port on the victim.
- *ttl*, time to live.
- *IpLen*, size of IP header in bytes.
- *DgmLen*, size of packet in bytes.
- *time*, time of occurrence.

For our honeypot experiment we used BRO-IDS. BRO creates a lot of different logs. We chose to use the signature log. This contains all the alerts Bro has raised from traffic matching its signatures. Some of the other logs Bro creates are: connection records, http, ftp and other application level protocols and a separate log for attacks detected from worms. Below is an example of a BRO alert.

```
1211608517.700684:SensitiveSignature:222.247.53.45:23505/tcp:
128.39.44.100:80/tcp:sid-ciac-7:222.247.53.45:
HXDEF 1.0-0.84 backdoor:GET http\://www.sciencedirect.com/
```

[4]Time To Live
[5]TOS: Type Of Service
[6]Datagram length: size of the packet

```
HTTP/1.1^M^JHost\: www.sciencedirect.com^M^JAccept\:
*/*^M^JPragma\: no-cache^M^JUser-Agent\: Mozilla/4.0 (compatible;...::
```

Every alert uses a single line instead of multiple lines compared to Snort, each of the attribute is separated by ":". Most of the logs are outdated and poorly documented.

```
<time>:<signature>:<source ip>:<src port/protocol>:<destination>:
<dst port/protocol>:<signature id>:<source ip, unknown?>:
<signature name>:<packet content>:<unknown>:
```

- <*time*>, a numeric value in seconds from, 1 January 1970.
- <*signature*>, What raised the alert, in the example above, a signature.
- <*source ip*>, origin of the packet.
- <*src port/protocol*>, port on origin's host.
- <*destination*>, destination of the packet
- <*dst port/protocol*>, port on destination.
- <*signature id*>, a unique id for the signature raising the alert
- <*source ip*>, duplicate entry of origin of the packet.
- <*sigature name*>, a short name for the signature.
- <*packet content*>, the content of the packet raising the alert
- <*unknown*>, unknown.

From the above attributes we chose the following attributes:

- *alert id*, used as event type.
- *time*, time of occurrence.
- *source ip*, attackers IP address.
- *destination*, IP address of the victim.
- *src port*, source port on the attacker.
- *dst port*, destination port on the victim.

A parser was written to extract the above attributes from each alert from both BRO and SNORT alert logs. Each alert would be grouped together with its attributes and written to a file ready for data mining. Event type was set to the unique signature identifier found in both Snort and Bro's alert log. The event sequence will be a 100% correct replication of the alert log in a format expected from our data mining algorithm.

In [27, 31] an extension to association and frequent episode mining is presented with the use axis attributes and reference attributes.

Axis These are attributes that are essential describing the data and must exist. Depending on the goal of data mining one or more axis attributes may be selected. With

axis attributes frequent episode first finds association about the axis attributes before it searches for other patterns.

Reference These attributes carry information describing the action of the event.

We have chosen to use the *alert id* and *time* as our axis attributes, with the rest are reference attributes. This gives a total of 7 reference attributes Event $A = ($*alert id*, *time*, reference attributes). Only the axis attributes are used during frequent episode discovery while the additional reference attributes are used to create rules and help with removing unwanted episodes, discussed in Section 3.3.

Example: With the following four SNORT alerts, taken from KDD Cup '99, Wednesday in week 4:

```
[**] [1:895:8] WEB-CGI redirect access [**]
[Classification: Attempted Information Leak] [Priority: 2]
03/31-15:03:15.323180 172.16.114.207:1025 -> 207.46.176.50:80
TCP TTL:64 TOS:0x0 ID:1498 IpLen:20 DgmLen:329 DF
***AP*** Seq: 0x1E189EF1  Ack: 0xFBF322FE  Win: 0x7D78  TcpLen: 20
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=2000-0382]

[**] [1:895:8] WEB-CGI redirect access [**]
[Classification: Attempted Information Leak] [Priority: 2]
03/31-15:03:16.982636 172.16.114.207:1025 -> 207.46.176.50:80
TCP TTL:63 TOS:0x0 ID:1498 IpLen:20 DgmLen:329 DF
***AP*** Seq: 0x1E189EF1  Ack: 0xFBF322FE  Win: 0x7D78  TcpLen: 20
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=2000-0382]

[**] [1:3151:4] FINGER / execution attempt [**]
[Classification: Attempted Information Leak] [Priority: 2]
03/31-15:04:13.010110 209.30.70.14:1026 -> 172.16.114.50:79
TCP TTL:62 TOS:0x0 ID:40 IpLen:20 DgmLen:43 DF
***AP*** Seq: 0xC2EABA52  Ack: 0xB808AC46  Win: 0x7D78  TcpLen: 20
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=2000-0915]

[**] [1:3151:4] FINGER / execution attempt [**]
[Classification: Attempted Information Leak] [Priority: 2]
03/31-15:04:13.238084 209.30.70.14:1027 -> 172.16.114.50:79
TCP TTL:62 TOS:0x0 ID:47 IpLen:20 DgmLen:43 DF
***AP*** Seq: 0x923436F1  Ack: 0x4A7EC29B  Win: 0x7D78  TcpLen: 20
[Xref => http://cve.mitre.org/cgi-bin/cvename.cgi?name=2000-0915]
```

The output from the alerts above is given below, which the format our data mining algorithm reads. Time is set to zero for the first alert in the sequence. This time will be used as a reference time and all alerts following will be offset in seconds from this. This also means all alerts happening within the same second will be set to occur at the same time. The order of the alerts are still preserved in the sequence even is if their accuracy is reduced to seconds.

```
[1:895:8] 0 172.16.114.207 207.46.176.50 1025 80 64 20 329
[1:895:8] 1 172.16.114.207 207.46.176.50 1025 80 64 20 329
[1:3151:4] 58 209.30.70.14 172.16.114.50 1026 79 62 20 43
[1:3151:4] 58 209.30.70.14 172.16.114.50 1026 79 62 20 43
```

3.3 Data mining

This phase is the core of our system, consisting of three steps:

- Finding frequent episodes
- Removing unwanted episodes
- Building attribute rules

We will present each step in an ordered way, Section 3.3.1 describes how frequent episodes work, Section 3.3.5 presents our algorithm for identifying and removing unwanted episodes, with the last step is explained in Section 3.3.6.

3.3.1 Frequent episodes

Frequent episode discovery is a descriptive data mining technique aimed at finding correlation between events in a long sequence of events as input. In our implementation every event within this sequence is the occurrence of an IDS alert. The type of event will be denoted by the ID of an IDS alert, explained in Section 3.2.1. Frequent episodes find events likely to occur together, called episodes. There are two important input to the algorithm, the frequency threshold and the window size. A confidence threshold can also be used but we have chosen not to use this due to some of the observations explained in more detail in Section 3.3.4.

By having a long sequence of events it is possible to find events that are likely to occur together. The frequent episodes algorithm finds both episode rules and association rules. In *WINEPI* an episode is a sequence of alerts occurring within a specified time window. For an episode to be considered frequent it has to occur several times, i.e. above a specified threshold. The difficult task of finding episodes in an event sequence is that there may be unrelated events between any of the events in an episode.

To ensure the algorithms were implemented correctly, they were run on small event sequences which could afterwards be checked manually. To further help, debugging information was also added to the algorithm to spot errors more easily.

By finding the association rules between the episodes, we can ignore episodes with a low confidence. By looking closer at episode $C \Rightarrow E \Rightarrow F$ above we find, it has the following sub episodes; C, $C \Rightarrow E$ and $E \Rightarrow F$ [7]. The confidence of $C \Rightarrow E \Rightarrow F$ occurring when C has occurred is $fr(C \Rightarrow E \Rightarrow F)/fr(C)$, which gives $0.10/0.59 = 0.17$.

The algorithms for finding frequent episodes [2] consists of two main parts. Finding episodes and calculating association rules between events and episode. The full implementation consists of four algorithms. The following steps are used to find frequent episodes in an event sequence:

1. Make a pass through the event sequence and find the frequency of each candidate, Algorithm 4.

[7] $C \Rightarrow F$ is also a sub episode, but is not used to create rules.

2. From the frequent episodes, suggest new candidates, Algorithm 3.
3. Repeat step 1-2 until there are no new candidates, Algorithm 2.
4. Build association rules, Algorithm 1.

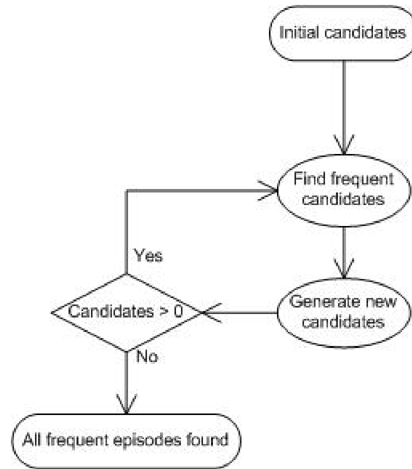


Figure 15: Flow diagram of frequent episode algorithm

Algorithm 1 is the entry point. As input it takes an event sequence **s** to find episodes in and a collection of base episodes. This collection will only contain episodes with a single event, these episodes must all be of the same size. These will be the smallest subepisode of any frequent episode found. By using only single events as an episode we ensure we will find every frequent episode. The purpose of Algorithm 1 is to generate all the association rules between subepisodes and their super episode. Assocciation rules tell the probability of the super episode occurring when the subepisode has occurred. This method of calculating accossiation rules only gives an estimate. This will be described in Section 3.3.4. A quick overview of the steps Algorithm 1 does is as follows:

1. Find all frequent episodes, Algorithm 2.
2. Find a super episode α and all its subepisodes β
3. Output the rule $\beta \rightarrow \alpha$ with confidence: $fr(\alpha)/fr(\beta)$
4. Repeat 2-3 for all episodes.

Algorithm 2 binds together the two core algorithms, Algorithm 3 and 4. It takes in a collection of candidates, which form the starting point for finding further larger episodes. These are run through Algorithm 4 to find which of them are frequent. The episodes found to be frequent are then passed on to Algorithm 3 to create new candidates. These new candidates will be one size bigger than the previews frequent episodes. This is done until there are no more candidates. A quick overview of the steps in Algorithm 2 is as follows:

1. Find all frequent episodes F_l from C_l in **s**, Algorithm 4.
2. Create new candidates C_{l+1} from F_l, Algorithm 3.
3. Repeat 1-2 until there are no new candidates, $C_{l+1} = \varphi$

3.3.2 Generation of candidate episodes

Algorithm 3 generates new candidates. In order to do so effectively, it uses the episodes found to be frequent by Algorithm 4. As input it takes a collection of frequent episodes, which has to be sorted and of the same size. This is important to be able to generate the candidates correctly. Episodes are sorted into *blocks*. Episodes within the same block share the same event except the last one. More formally, if $F_l[i]$ and $F_l[j]$ of size *l* share *l-1* events, then for $i \leq k \leq j$ they also share the same events. All events from $[i, j]$ will be in the same block. This is to make sure we generate as few candidates as possible. Line 9-10 of Algorithm 3 creates all permutations between the episodes within the same block. The for loop from 12-15 in Algorithm 3 tests all sub episodes of the new candidate, and checks if they are frequent. They all have to exist in order for the new candidate to be frequent. A episode cannot be frequent if any of its sub episodes are not (see the Lemma stated in Section 2.1.3).

Example: Given the frequent episodes:

- A
- B
- C
- $A \Rightarrow B$, block 1
- $A \Rightarrow C$, block 1
- $B \Rightarrow B$, block 2

Only $A \Rightarrow B$, $A \Rightarrow C$ and $B \Rightarrow B$ will be used to create new candidates. Frequent episode of size l are used to create candidates of size $l + 1$, i.e. A, B and C were used to create candidates of size 2, i.e. $A \Rightarrow B$, $A \Rightarrow C$ and $B \Rightarrow B$ (more was created but only these were found to be frequent).

$A \Rightarrow C$ and $A \Rightarrow B$ are in the same block, and $B \Rightarrow B$ is in a block for itself. This leads to the following new candidate: $A \Rightarrow B \Rightarrow B$ from block 1, the second block ($B \Rightarrow B$) can only suggest $B \Rightarrow B \Rightarrow B$ as a new candidate, since there are no other episodes to combine with within the same block. This goes back to the lemma for sub episodes in Section 2.1.3. $B \Rightarrow B \Rightarrow C$ cannot be frequent since $B \Rightarrow C$ is not frequent. The same goes for $A \Rightarrow B \Rightarrow C$, $A \Rightarrow C \Rightarrow C$, $A \Rightarrow C \Rightarrow B$, none of these can be frequent since some of their sub episodes are not frequent.

One thing to take notice of is that Algorithm 3 calculates the block each episode belongs to, but these values become incorrect once one episode is removed. Unless all candidates are frequent after they have been run through Algorithm 4, which only return the frequent episodes, it is important to recalculate the blocks for the episodes. The reason Algorithm 3 is calculating these blocks, is that it is often very time consuming to make a pass through the event sequence. Generating candidates are on the other hand usually very fast compared to finding frequent episodes and it can therefor be desirable to make several passes at generating candidates. This will not miss any episodes as it only assumes all the previews candidates are frequent and used to make candidates, which are frequent will be found with a run through Algorithm 4.

1. Find next block of episodes.
2. Create all possible permutations between episodes within the block.
3. Test if all sub episodes of each new candidate is (C_{l+1}), is in F_l ($\beta \in \alpha \in C_{l+1}$ and $\beta \in F_l$).
4. Repeat 1-3 for each block.

3.3.3 Finding frequent episodes

Algorithm 4 returns episodes from a collection of candidates (C) which are frequent in event sequence **s**. There are no restrictions on C as it does not have to be sorted in any way or only contain episode of similar size. To find the frequency of each episode it moves a window over the sequence and tests if each episode is present inside the window. Every episode can only occur once in each window. By counting the number of occurrences of an episode it is considered frequent if it is above a user defined threshold (min_fr).

Algorithm 4 takes four inputs; C, **s**, win_size, min_fr. The total number of windows in sequence **s** will be $\mathbf{s}.T_e$ - $\mathbf{s}.T_s$ + win_size. The first window is at $\mathbf{s}.T_s$ - win_size +1, while the last window is at $\mathbf{s}.T_e$ + win_size -1. This is to ensure every event is within the window an equal amount of time.

1. Set window at $\mathbf{s}.T_s$ - win_size +1.
2. Check each episode if they exist inside the window.
3. Move window one time unit.
4. Repeat 2 while window < ($\mathbf{s}.T_e$ + win_size -1).
5. Calculate frequency of each episode.
6. Return only frequent episodes.

Example:

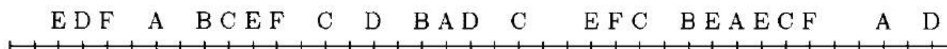


Figure 16: Sequence of events [2]

The sequence in Figure 16 with a frequency threshold of 0.1 and a window size of 5 would give the output below. This shows the episodes found for each pass along with the number of occurrences and how frequent they are in the sequence. Each episode can only occur once in every window. By manually moving a window over the sequence the occurrence of every episode could be counted and checked.

```
Frequent episodes (size 1):
 E  - fr(22): 0.536585
 D  - fr(19): 0.463415
 F  - fr(20): 0.487805
 A  - fr(20): 0.487805
 B  - fr(15): 0.365854
 C  - fr(24): 0.585366
Frequent episodes (size 2):
 E -> D  - fr(4): 0.097561
```

```
 E -> F  - fr(14): 0.341463
 E -> A  - fr(5): 0.121951
 E -> C  - fr(9): 0.219512
 D -> F  - fr(4): 0.097561
 D -> A  - fr(4): 0.097561
 F -> A  - fr(5): 0.121951
 F -> C  - fr(7): 0.170732
 A -> E  - fr(5): 0.121951
 A -> D  - fr(7): 0.170732
 A -> C  - fr(7): 0.170732
 B -> E  - fr(7): 0.170732
 B -> A  - fr(7): 0.170732
 B -> C  - fr(6): 0.146341
 C -> E  - fr(8): 0.195122
 C -> F  - fr(8): 0.195122
 C -> B  - fr(4): 0.097561
Frequent episodes (size 3):
 E -> F -> C  - fr(5): 0.121951
 C -> E -> F  - fr(4): 0.097561
```

With only six different event types and a short sequence with a time span of 38 time units, it finds a total of 25 episodes. This is partly from the large window size compared to the data set, but it is a well know problem with episode rules a lot of irrelevant and redundant episodes are found [1].

By looking closer at episode $E \Rightarrow D$ we can check its frequency and occurrence in the following way. From the sequence in Figure 16 there are only two places where E and D occur closely, $(E,2)(D,3)$ and at $(E,10)(D,15)$. The last one is just too far apart, just as D enters the window, E leaves it. This only leaves the windows below where E and D are present within the window at the same time.

				E
			E	D
		E	D	F
	E	D	F	
E	D	F		A
D	F		A	

Table 1: Window content as it is moved over where E and D are close

From the Table 1 we can verify the frequency of the episode. With it occurring in four windows within the sequence containing a total of 42 different window, by applying these values to Equation 2.1 we get the following:

$$fr(\alpha, S, |W|) = \frac{4}{42} = 0.097 \tag{3.1}$$

3.3.4 WINEPI Observation

Due to the way *WINEPI* calculates its confidence value it can only be considered an estimate. Two events not occurring at the same time can never have a confidence of

100%, even if they always occur together. As an example take the sequence in Figure 17, the content of a window sliding over this sequence is shown in Table 2.

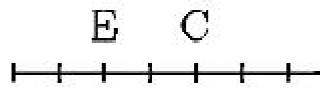


Figure 17: Sequence of events

			E
		E	
	E		C
E		C	
	C		
C			

Table 2: Window content as its moved over the sequence in Figure 17

It becomes apparent that E (and C) will appear in two windows more than E ⇒ C making them more frequent. The confidence value will therefore be an estimate but never a correct value. The more E ⇒ C occurs in the sequence, the less accurate the confidence value will get. This raises another observation, e.g. if we have an event that triggers another event 20 time units later, we might expect that this gets picked up as frequent if we use a window size of 20. Now since both only occur within our window for one step before one of them is shifted out again, this makes it unlikely they are found. To compensate for this problem the window size could be increased to 40 where the events would occur within 20 windows before disappearing again. From this we can denote, the closer two events occur together the higher is the probability they become frequent, finding episodes with events further apart then half the window size therefore unlikely. With this in mind and the knowledge that frequent episode mining often leads to a vast amount of patterns [4] we developed an additional step to compensate for these observations.

Window size is not to be confused with the maximum size an episode can be. The size of an episode does not tell how far apart the events are only they are likely to occur together. Several events might occur at the same time and if a episode with this is found with high burst of event one might get episodes of more events than the window size, but the first and last event can never be further apart then the 'window size' in time units.

3.3.5 Removing unwanted episodes

As described in Section 3.3.2 to find episodes efficiently all subepisodes of an episode have to be frequent for it to be considered. Searching for episodes this way ensures every possible frequent episode is found. This creates a lot of unwanted or unnecessary episodes as a result. With an episode, e.g. A ⇒ B ⇒ C with a confidence of 100% between each event, all sub episodes of this episode are unwanted for any other purpose then generating the association rule stating once A has occured B and C will follow.

To find these unwanted episodes we developed our own algorithm 5 to find and remove these. As input it takes the same window size and sequence used in *WINEPI*, along with all the frequent episodes found. It is important to notice that the collection of frequent episodes is sorted by size, where F[1] contains the largest episode while smallest

resides at $F[|F|]$. It uses a sliding window over the sequence matching episodes against the content of the window. Only episodes starting with the same event as the last event within the window are used ($\alpha[1] = W[1]$). This is done in a descending order where the largest episodes are tried to first. Matching against the largest first is important so a match against one of its sub episodes is not achieved instead. Using the last event within the window ensures every episode has equal chance to get a match.

The events within the window matching the episode will be removed before the window is moved another step, to ensure an event can only belong to one episode, enhancing the cause and consequence between events. We want to find episodes where the cause of one event occurring triggers the next event. These episodes we consider "final" episodes. Only episodes actually used to remove alerts from the sequence will be treated as "final". These are the ones we build rules from. As discussed earlier the confidence value calculated by *WINEPI* is only an estimate. Along with finding "final" episodes our algorithm also finds a new confidence value for each episode.

In our Algorithm 5 two variables are used, final_hit and sub_hit. When an episode α is found inside a window, α.final_hit and α.sub_hit are increased, for all sub episodes of α, only sub_hit is increased (β.sub_hit). It is important to use separate values. Only those episodes, which have a final_hit value higher that 1 will indicate these episodes were found within the sequence, and not part of the subepisodes used to find the final episode. The confidence values between episodes can be found the same way as in Algorithm 1 from *WINEPI*, were frequency $fr(\alpha) = \alpha.sub_hit + \alpha.final_hit$.

$$conf(\beta \Rightarrow \alpha) = \frac{fr(\alpha)}{fr(\beta)} = \frac{\alpha.sub_hit + \alpha.final_hit}{\beta.sub_hit + \beta.final_hit} \tag{3.2}$$

One thing that needs careful consideration is when there exist several alerts within the window matching the episode. Getting the correct alerts matched against an episode is important when we use each occurrence of an episode to create additional attribute rules. This is explained in further detail in Section 3.3.6. From the alerts within the window, we need to determine which is the best alert to match against the episode. To do this we used a penalty function returning a value signifying how equal two alerts are based on their attributes. It tests the attributes from both events against each other. Each attribute is assigned a value signifying the importance of it. If the two attributes are equal a value of zero is returned while a value greater than zero is returned if the alert attributes are not equal. We used the following values for our penalty function:

$$penalty(A, B) = \begin{cases} 10 & \text{if A.source} \neq \text{B.source} \\ 10 & \text{if A.destination} \neq \text{B.destination} \\ 3 & \text{if A.dest_port} \neq \text{B.dest_port} \\ 3 & \text{if A.src_port} \neq \text{B.src_port} \\ 1 & \text{if A.ttl} \neq \text{B.ttl} \\ 1 & \text{if A.iplen} \neq \text{B.iplen} \\ 1 & \text{if A.dgmlen} \neq \text{B.dgmlen} \end{cases} \tag{3.3}$$

$$\sum_{i=1}^{n} penalty(A.a_i, B.a_i) \tag{3.4}$$

IP address of the attacker or the victim are two attributes expected to have a strong relations between each alert. We have therefore chosen to give these two attributes a high penalty for not being equal. While port address has lower number, with the remaining

attributes being even less important, all values returned from each attribute are added together to form the final value for the two alerts. The lower this score is the better match the two alerts are, with zero being a perfect match. The steps of our algorithm can be summarized as follows:

1. Start window T_s.
2. Find all episodes with the same starting event as the window ($\alpha[1] = W[1]$).
3. Test each episode for a match against the window, starting with the largest episodes.
4. If a matching episode is found, remove matching events from the window.
5. Repeat 2-4 until window is at T_e.

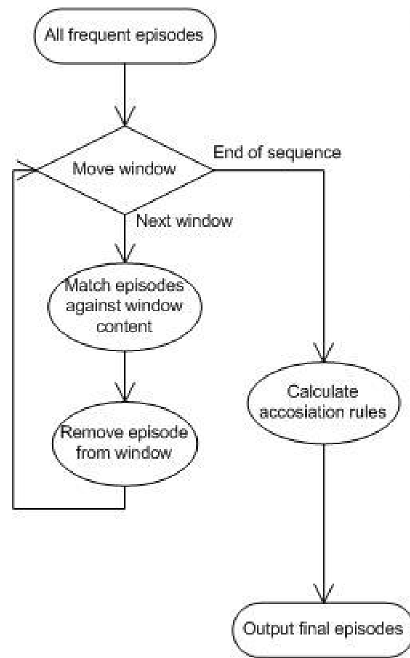


Figure 18: Flow diagram of the episode removal algorithm

Example: Table 3 shows the content of a window with two occurrences of the same events B_1 and B_2. Both are of the same event type, but their attributes are not. We have an episode $\alpha : (A \Rightarrow B)$, with A matched against the last event within the window. We need to determine which B event within the window provides the best match.

A	D	B_1	F	B_2	F

Table 3: Content of a window

Both events are matched against A ($x = penalty(A, B_i)$). With the attributes given in Table 4 the two following Equations 3.5 and 3.6:

$$\sum_{i=1}^{7} penalty(A.a_i, B_1.a_i) = 0 + 10 + 3 + 3 + 1 + 0 + 1 = 18 \tag{3.5}$$

	A	B_1	B_2
Source	123.123.123.1	123.123.123.1	123.123.123.1
Destination	111.111.111.111	222.222.222.222	111.111.111.111
Source port	1	2	3
Destination port	80	22	80
TTL	36	65	78
IP length	20	20	20
Datagram length	346	42	456

Table 4: Attributes of events A, B_1 and B_2.

$$\sum_{i=1}^{7} penalty(A.a_i, B_2.a_i) = 0 + 0 + 3 + 0 + 1 + 0 + 1 = 5 \quad (3.6)$$

From Equation 3.5 and 3.6, B_2 gets the lowest score and provides the best match for our episode. From the calculations, events A and B_2 will be considered an occurrence of the episode $A \Rightarrow B$ for further processing.

3.3.6 Attribute Rules

After we have found "final" episodes, the last step is to create attribute rules for the episodes discovered. We create both inter- and intra-episode attribute rules. These rules are created from each occurrence of the episode within the alert sequence. Inter-episode finds rules between the attributes of the same event from different occurrences of the episodes. These rules will therefore contain set values, e.g. $\alpha[1]$.ttl = 123. These rules state that the first event in episode α and its attribute 'ttl' should always contain the value 123 for every occurrence of this episode.

Intra episode rules find between the attributes of the alerts within the same occurrence of an episode, i.e. $\alpha[1].a_i = \alpha[2].a_i$. Intra episode rules help us find relations between the attributes of the alerts within a single occurrence of an episode. $\alpha[1].a_i = \alpha[2].a_i$ will tell us that both the first and second alert have the same value for this attribute. Attribute rules are mined with a 100% confidence as they are used to give a stricter description of the episode, without removing any of the found episodes.

Example: With an episode $A \Rightarrow B \Rightarrow C$ and it having two occurrences specified in Table 5 and 3.3.6.

	A	B	C
Source	123.123.123.1	123.123.123.1	123.123.123.123
Destination	222.222.222.222	222.222.222.222	222.222.222.222
Source port	1	2	3
Destination port	80	80	80
TTL	36	65	78
IP length	20	20	20
Datagram length	346	42	456

Table 5: Attributes of events A, B_1 and B_2.

1. Build inter-episode attribute rules.
2. Build intra-episode attribute rules.

	A	B	C
Source	33.33.33.33	33.33.33.33	33.33.33.33
Destination	222.222.222.222	222.222.222.222	222.222.222.222
Source port	9	8	7
Destination port	80	80	80
TTL	53	34	73
IP length	20	20	20
Datagram length	54	355	435

3. Test each episode against each attribute rule.
4. Output passing rules.

The following attribute rules will be made:

```
IF      A
THEN    B
        C

IF      A
        B
THEN    C

A.dst_ip = B.dst_ip = C.dst_ip
A.src_ip = B.src_ip = C.src_ip
A.dst_port = B.dst_port = C.dst_port
A.iplen = B.iplen = C.iplen

A.dst_ip = 222.222.222.222
B.dst_ip = 222.222.222.222
C.dst_ip = 222.222.222.222
A.dst_port = 80
B.dst_port = 80
C.dst_port = 80
A.iplen = 20
B.iplen = 20
C.iplen = 20
```

The above is an example of what a data mined rule might look like. Note that even if A and B occur without C, this does not satisfy the rule. To summarize how this is read:

- Given A has occurred, there is a 60% chance B and C will follow.
- Given A and B has occurred, there is a 70% chance C will follow.
- The 'dst' attribute of event A, B and C all have the same value.
- A.dgmlen = 300 and B.dgmlen = 50, tell the fixed values of attribute 'dgmlen' for event A and B.

3.4 Analysis

This is the last phase in our data mining algorithm. This does not do any more processing of the rules, but only presents the final rules to an analyst to make the final decision if the rule should be applied or not. This step is quite important and the analyst should be through with the investigation of the rules presented. Accepting rules not modeling false positives can have a severe effect on the security of the system. This will be further discussed in Section 5.4.

3.5 Applying filter

After the final rules have been created and accepted by the analyst, these can be used to filter the alert log. The alert log is searched for alerts matching the rules requirements and conditions. This is done in much the same way we removed unwanted episodes. The final decision for a rule match cannot be taken before the whole window has passed. Even if the rule finds all alerts within a window it needs to be satisfied they can belong to other similar rules, and an alert can only be used as a match against a single rule. Take the two episodes: $\alpha : (\mathsf{E} \Rightarrow \mathsf{F})$ and $\gamma : (\mathsf{C} \Rightarrow \mathsf{F})$ when alert F occurs, and given both E and C has occurred earlier, we cannot tell if it belongs to α or γ. Both rules will have a match against these alerts, given attribute rules are fulfilled too, but only one rule can be used to match against F.

Because of this the final decision for a rule match cannot be taken before it is at the end of the window ($W[1]$). Some alerts might have been removed from earlier rule matches. In the case where several alerts match up against a rule, we use the same penalty function described in Section 3.3.5 to find the best alert for our episode.

Example:

- Red: $\alpha(\mathsf{E} \Rightarrow \mathsf{F})$
- Blue: $\beta(\mathsf{B} \Rightarrow \mathsf{A})$
- Green: $\gamma(\mathsf{C} \Rightarrow \mathsf{F})$

With the three simple rules above, Figure 19 shows how an alert sequence is filtered according to these rules.

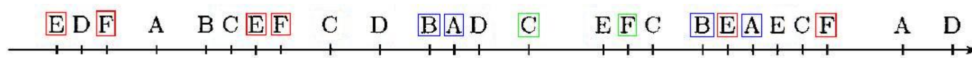


Figure 19: Filtered alert sequence

Notice how the Green rule gets mapped against $\mathsf{C} \Rightarrow \mathsf{F}$ and not the red rule, since C occurred before E. This is the main reason the final decision for a rule cannot be take before the end of the window.

3.6 Formal definition of algorithms

Algorithm 1 Rules algorithm

1: * Find frequent episodes (Algorithm 2): *\
2: compute $F(\mathbf{s}, \mathit{win}, \mathit{min_fr})$;
3: * Generate rules: *\
4: **for** all $\alpha \in F(\mathbf{s}, \text{win}, \text{min_fr})$ **do**
5:   **for** all $\beta \in \alpha$ **do**
6:     **if** $\text{fr}(\alpha)/\text{fr}(\beta) \geq \text{min_conf}$ **then**
7:       output the rule $\beta \rightarrow \alpha$ and the confidence $\text{fr}(\alpha)/\text{fr}(\beta)$
8:     **end if**
9:   **end for**
10: **end for**

Algorithm 2 Main algorithm

1: $C_1 = \{\alpha \in \epsilon \mid |\alpha| = 1\}$;
2: $l = 1$;
3: **while** $C_l \neq \emptyset$ **do**
4:   * Database pass (Algorithms 4): *\
5:   compute $F_l = \alpha \in C_l \mid fr(\alpha, s, win) \geq min_fr$;
6:   $l = l + 1$;
7:   * Candidate generation (Algorithm 3): *\
8:   compute $C_l = \{\alpha \in \epsilon \mid |\alpha| = l$ and for all $\beta \in \epsilon$ such that $\beta \prec \alpha$ and $|\beta| < l$ we have $\beta \in F_{|\beta|}\}$;
9: **end while**

Algorithm 3 Create new candidates

1: $C_{l+1} = \emptyset$;
2: $k = 0$;
3: **if** $l = 1$ **then**
4:     **for** $h = 1$ to $|F_l|$ **do**
5:         $F_l.block_start[h] = 1$;
6:     **end for**
7: **end if**
8:
9: **for** $i = 1$ to $|F_l|$ **do**
10:     current_block_start = k + 1;
11:     **for** $j = i; F_l.block_start[j] = F_l.block_start[i]; j = j + 1$ **do**
12:         $\backslash *$ $F_l[i]$ and $F_l[j]$ have l-1 first event types in common, build a potential candidate α as their combination: $* \backslash$
13:         **for** $x = 1$ to l **do**
14:             $\alpha[l + 1] = F_l[j][x]$;
15:         **end for**
16:         $\alpha[l + 1] = F_l[j][l]$;
17:         $\backslash *$ Build and test sub episodes β that do not contain $\alpha[y]$: $* \backslash$
18:         **for** $y = 1$ to $l - 1$ **do**
19:             **for** $x = 1$ to $y - 1$ **do**
20:                 $\beta[x] = \alpha[x]$;
21:             **end for**
22:             **for** $x = y$ to l **do**
23:                 $\beta[x] = \alpha[x + 1]$;
24:             **end for**
25:             **if** $\beta \notin F_l$ **then**
26:                 continue with next j at line 11;
27:             **end if**
28:         **end for**
29:         $\backslash *$ All sub episodes are in F_l, store α as candidate: $* \backslash$
30:         k = k + 1;
31:         $C_{l+1}[k] = \alpha$;
32:         $C_{l+1}.block_start[k] = current_block_start$;
33:     **end for**
34: **end for**
35: ouput C_{l+1};

Algorithm 4 Finding frequent episodes

```
1:  \∗ Initialization: ∗\
2:  for each α ∈ C do
3:     for each A ∈ α do
4:        A.count = 0;
5:        for i = 1 to |α| do
6:           contains(A, i) = ∅;
7:        end for
8:     end for
9:  end for
10:
11: for each α ∈ C do
12:    for each A ∈ α do
13:       a = number of events of type A in α;
14:       contains(A, a) = contains(A, a) ∈ α
15:    end for
16:    α.event_count = 0;
17:    α.freq_count = 0;
18: end for
19: \∗ Recognition: ∗\
20: for start = T_s − win + 1 to T_e do
21:    \∗ Bring in new events to the window: ∗\
22:    for all events(A,t) in s such that t = start + win − 1 do
23:       A.count = A.count + 1;
24:       for each α ∈ contains(A, A.count) do
25:          α.event_count = α.event_count + A.count;
26:          if α.event_count = |α| then
27:             α.in_window = start;
28:          end if
29:       end for
30:    end for
31:    \∗ Drop out old events from the window: ∗\
32:    for all events(A, t) in s such that t = start − 1 do
33:       for each α ∈ contains(A, A.count) do
34:          if α.event_count = |α| then
35:             α.freq_count = α.freq_count − α.in_window + start;
36:          end if
37:          α.event_count = α.event_count − A.count;
38:       end for
39:       A.count = A.count − 1;
40:    end for
41: end for
42: \∗ Output: ∗\
43: for all α ∈ C do
44:    if α.freq_count/(T_e − T_s + win) ≥ min_fr then
45:       output α;
46:    end if
47: end for
```

Algorithm 5 Remove unwanted episodes

```
1: \* Initialization: *\
2: for each α ∈ F do
3:    α.hit_count = 0;
4:    α.sub_hit = 0;
5: end for
6: for start = T_s − win + 1 to T_e do
7:    \* Bring in new events to the window: *\
8:    for each events(A,t) in s such that t = start + win − 1 do
9:       W.add(A);
10:   end for
11:   \* Drop out old events from the window: *\
12:   for all events(A, t) in s such that t = start − 1 do
13:      W.remove(A);
14:   end for
15:   for each A ∈ W[1] do
16:      for each α ∈ F where α[1] = W[1] do
17:         if α ∈ W then
18:            for each β ∈ α do
19:               β.sub_hit = β.sub_hit + 1
20:            end for
21:            for each A ∈ F do
22:               W.remove(A);
23:            end for
24:            α.sub_hit = α.sub_hit + 1
25:            α.hit_count = α.hit_count + 1;
26:         end if
27:      end for
28:   end for
29: end for
30: \* Output: *\
31: for all α ∈ F do
32:    if α.hit_count ≥ min_fr then
33:       output α;
34:    end if
35: end for
```

Algorithm 6 Build rules

```
1: for each rule R do
2:    \* Build intra-episode rules: *\
3:    for each occurrence of α in s  do
4:       for each α.a_i do
5:          compute α.a_i = α.a_j;
6:       end for
7:    end for
8:    \* Build inter-episode rules: *\
9:    for each α.a_i do
10:      for each occurrence of α.a_i in s do
11:         compute α.a_i = γ.a_i = CONSTANT, where α ≡ γ;
12:      end for
13:   end for
14: end for
```

Algorithm 7 System algorithm

1: * Find frequent episodes: *\
2: compute $F(\mathbf{s}, win, min_fr)$;
3: * Remove unwanted episodes: *\
4: remove $\beta \in \alpha \exists F(\mathbf{s}, win, min_fr)$;
5: * Build intra-episode rules: *\
6: compute $\alpha \exists F(\mathbf{s}, win, min_fr) \; | \gamma \equiv \alpha | \alpha.a_i = \gamma.a_i$;
7: * Build inter-episode rules: *\
8: compute $\alpha \exists F(\mathbf{s}, win, min_fr) \; | \gamma \equiv \alpha | \alpha.a_i = \gamma.a_i = CONSTANT$;

4 Data gathering

The purpose of our experiment is to investigate if there is a possibility to find correlation between several IDS alerts as a frequent episode. As presented in Chapter 3 we will use data mining techniques trying to find patterns in IDS alert logs. From our early research we found frequent episode discovery had a known problem with finding lots of redundant and irrelevant episodes [28, 29, 30]. With our experiments we wanted to evaluate both how effective our filter method is and how well we removed unwanted episodes.

Both intrusion detection systems have been used with default settings without adjusting or customizing network specific settings or rules in order to demonstrate the performance of our system in reducing the amount of false positives and therefore reducing time-consuming IDS tuning.

There may be several advantages with finding patterns in alarm logs can be several. By finding groups of alerts occurring together can have a significant impact on the number of alerts an IDS analyst has to check. These groups can be flagged according to the analyst's desire to either discard or keep them.

4.1 Experimental setup

When it comes to testing intrusion detection systems and generating alerts, this can be done in several ways. Most IDS tests can be put into four different categories [32]: no background traffic, using real traffic, using sanitized traffic or using simulated traffic. Which one is the best depends on what one is looking for, since they all have distinct advantages and disadvantages.

No background traffic The IDS is set up on a network with no traffic on it, and attacks are launched to see if the IDS can detect them. This approach will tell nothing about false positives of IDS. This can be used to verify which attacks an IDS can detect and that it labels them correctly. This is probably the easiest and most cost effective way of testing an IDS system. This method assumes that background traffic does not affect an IDS's ability to detect intrusions. While this might hold true for low rates of traffic, when approaching the IDS's resource limit for processing this is not true any more.

Testing using real traffic One could decide to test the intrusion detection system by deploying it on a live network, by injecting attacks into the network. We will see which attacks it detects and get a rudimentary false positive and false negative values. Problem is that there might exist other attacks on the network from real threats taking place. So any true positives have to be investigated in order to se if they originate form the injected attacks and if they do not, they have to be checked if they originate from a real attack, otherwise it is an false positive. This task can quickly become daunting and very time consuming to complete. It might even miss the real attacks completely and one can therefore never get a ROC[1] from this kind

[1] ROC: Receiver operating characteristic

of testing, as going through all the data and verifying it for attacks is impossible. Since the test is performed with live data, it makes it impossible to reproduce the same on a different IDS making it hard to compare systems tested at different times. This holds for a less scientific evaluation, but evaluates the IDS in the environment it is meant to be running.

Testing using sanitized traffic Using real background traffic makes it impossible to log this and replay it later due to privacy issues. Research has been done to remove sensitive or personal information [33]. By recording traffic and then applying sanititation algorithms, it is possible to inject attacks into the stream. There are still problems with this approach. The sanitation might remove too much leaving an unrealistic view of the environment. Working on wast amount of data which is impossible to go over and check manually, the process might leave traces of sensitive data. Attacks injected into a stream of normal operating network might cause other problems too. With a successful attack on a computer disabling the prerecorded traffic will still run, with normal traffic flowing to and from the attacked computer in an unnatural way, from the earlier simulated background traffic.

Testing by generating traffic on a testbed network This is the most common approach by creating a test network where the IDS can be deployed and tested [34, 35, 36]. Even though this model requires complex traffic generators that model the type of traffic, the main advantage here is that one is sure that the traffic does not contain any additional attacks. Logging and replaying the data at a later point is also possible since it will not contain any personal or sensitive data. This would be one of the best methods to test both true positives and false positives of an IDS system. There are some drawbacks to setting up such a system. Creating a correct model for the IDS to be tested in would require a lot of time and effort and would probably be the most expensive. Simulating high bandwidth environment could prove to be a challenge. Both the model for which the traffic generators generate traffic and the way the network is structured need to be carefully considered, since different types of networks have different traffic properties.

To generate alerts we have chosen to go with a variant of both the first and the last option, choosing to use both a real network traffic and an artificial data set. KDD Cup '99 is a common and well documented data set, often used in academic environments and is known to those needing to test IDS. When using live network traffic it is difficult to make sure the experiment is fully repeatable due to restrictions on traffic capture, as it can contain sensitive information, but it provides a real test scenario with modern traffic. Deploying on a real network verifying if alerts are false positives or true positives can become a time consuming task. With a known dataset which has known attacks, and attack free data set is of great help.

4.1.1 KDD Cup '99

There exist several data sets for benchmarking IDS systems. Unfortunately few of these are commonly available. The most commonly used is published by Lincoln laboratory and was created in co-operation with DARPA. This data set simulates data traffic on a military air-base for five weeks. The data set is split in two parts. The first three weeks contain training data aimed at anomaly detection IDS or other systems relaying on training data.

The two last ones are used for performance testing. The set contains 58 different attacks which can be categorized into 4 types:

- **Denial of service** (DoS) attack tries to deny a service provided by a system. This is commonly done by a SYN-Flood, flooding the victim with more connections that it can handle.
- **Probe attacks** tries to gain an overview over a network and it is structure, by finding live hosts and the port they respond to.
- **Remote to local** (R2L) attacks are attempts at gaining local access on the computer.
- **Local to root** (L2R) are attacks trying to give the attacker root or administrator privileges on the host.

John McHugh's paper[13, 37] looks at some of the shortcomings of the benchmark, but also adds that he feels this is the only IDS benchmark worthy of such a paper.

- Simulated normal usage is said to be similar to the normal usage of a real network. However, no further data are provided to prove this claim except a few things such as average number of words in a mail (which remains classified). This also raises many questions about how the normal usage is modeled.
- Data rate in the recorded traffic is suspiciously low. It takes only a few hours to replay traffic, which is supposed to be from thousands of hosts.
- Internet traffic is not well behaved.
- Unlike most real networks, all the hosts on the simulated network are connected to the same hub.
- ROC curves are not the best way to evaluate an IDS. It is useful if the IDS can produce different false alarms and detection rates on different configuration. In some cases the ROC will only contain one point.

There exist two versions of KDD Cup data set, the full tcpdump of all network traffic, with a total of 4 GiB of data, and a 10% reduced data set where attacks are labeled. We opted to use the full KDD Cup '99 data set, as SNORT has no way of reading the reduced data set. In total it contains 5 weeks of data with the three first being various training sets for intrusion systems relaying on this. There are 201 instances of 56 different attacks distributed throughout these two weeks. Traffic is split into 10 files for each week, two files for every day from Monday to Friday. This data set does not model traffic on weekends. Each day has two files, one with traffic originating from outside the network, while the other has internal traffic. One day in week 4 is missing its inside traffic, this will give some repercussions, if this day should be used at all. Only SNORT was used on this data set as BRO was not able to run offline signature analysis.

4.1.2 Honeynet

Due to KDD Cup being a data set for testing IDSs with artificial traffic, with some of the limitations of the old KDD Cup data set, a test with newer traffic was also necessary. With KDD Cup simulating an air force base, it hardly contains what can be considered normal network traffic. Some changes in traffic patterns are highly expected compared with the 10-years old traffic of KDD Cup data set, especially with the huge growth Internet has experienced, and with the activity from bot, viruses and worms.

For the purpose of gathering alerts from real network traffic, a small network of honeypots was deployed. We deployed both Bro and Snort on this network, as they were running on live traffic. We chose to use high interaction honeypots to be able to simulate as closely as possible a small network and letting the attacks occur on real hosts with running services. With the few months available for gathering alerts and the low profile of the network no real attacks were expected other than automated attacks from bots and worms. We did not use any application to generate attacks on these as the purpose of this setup is to see if internet traffic originating from worm or scan-bots can become frequent enough in an IDS log to be picked up by our system.

Honeypots are computer setup for the purpose of attracting attackers. They have no production value to the network, therefore all traffic going to and from a honeypot is considered highly suspicious. Honeypots should allow the attacker to continue their activities for as long as possible without compromising other parts of the network, but at the same time limit the amount of damage they can do. Honeypots can give valuable information on new attacks and provide an early warning. Honeypots can be split into two different types [12]:

- *Low interaction*, these honeypots will only emulate services. The computer will only appear to run the defined services and OS. Responses will usually be with predefined values and depend on the how well the service is emulated. These honeypots require little maintenance.
- *High interaction*, are setup with the actual services running. With a full implementation of services on the honeypot it will be vulnerable to attacks.

With the lack of a full implementation of the service, low interaction honeypots might provide less detailed information about attacks compared to a high interaction honeypot, due to the attacker might notice the responses from the victim is wrong, and only partly executing the attack. With the ease of deploying low interaction honeypots they are often used to provide an early warning of attacks, and wasting the time of the attacker. Since they are not vulnerable to attacks on their emulated services they can be employed within the network itself. There have been ways to detect low interaction honeypots [38, 39, 40]. Examples of low interaction honeypots are: Honeyd [41] and Netbait [42].

High interaction honeypots are hosts containing full implementation of services. This means these are fully vulnerable to attacks. These will respond in a way the attacker expects, making it even possible for the attacker to seize controll of the host. With an attacker actually being able to gain controll over the honeypot, a lot more consideration has to be taken when deploying and maintaining the hosts. The attacker should not be able to use the computer for any illegal actions, or help him gain further leverage to

attack the network. With a high interaction honeypot more details about attacks can be gathered. An example of high interaction honeypot is Honeywall [43].

4.1.3 Honeywall

Honeywall in itself is not a honeypot, but provides the tools necessary to set up high interaction honeypots. A high interaction honeypot is just a normal computer along with some installed software for different services, e.g. HTTP, SMTP and FTP. Honeywall uses three network interfaces, two of them are bridged, and just forwards traffic between them. The third one is used for management and setup. Honeywall logs all traffic going through it with SNORT.

Since Honeywall acts as a bridge it is virtually impossible to trace it. This does not mean it is not detectable but currently there is no known way of detecting it. The most important tools honeywall provides is to limit the use an attacker might have of compromising one of the hosts deployed as a honeypot. It acts as a strict firewall for all registered honeypots, allowing only a set amount of outgoing connections from a honeypot to occur within a set time limit. The ability to monitor SSL connection is also possible through Sebek, which is integrated into Honeywall. Sebek requires a kernel driver to be installed on all monitored hosts (i.e. honeypots). This was not installed as we did not setup any services requiring SSL.

To setup our network we used three separate computers. One with Honeywall, BRO and last one acted as honeypot with Windows XP installed as OS. BRO was installed with latest stable version, v1.2.1 while SNORT used version 2.8 with latest updated rule set for registered users as of 5th March 2008.

Windows XP	Ubuntu	Debian	IDS
Apache			Snort
ServU-FTP	VNC Server		Bro
MS-SQL			

Table 6: Overview of installed software

To allow for more client on the network VMware was installed on the honeypot. VMware was used to run two linux distributions, Ubuntu 8.4 and Debian (etch).

4.2 Experiments

To help answer our research question we used three experiments. In this section we provide a short overview of the experiments and our goal for each.

4.2.1 Experiment 1: Initial test

Our system relies on two input variables: *frequency threshold* and *window size*. In this experiment we wanted to get a better understanding of how these two variables affect the rules generated by our system. These tests were done on one of the attack free weeks in KDD Cup. The frequency threshold was tested in ranges of 0.02 to 0.003, while window size ranged from 30 to 90. During this experiment tests of how effective the algorithm for removing unwanted episodes was also investigated.

4.2.2 Experiment 2: True positive rules from honeynet

This experiment setup a small honeynet with the purpose of generating alerts from malicious internet traffic. Our main goal for this experiment is to discover how frequent many

of automated attack from internet can become. All rules generated from these alert logs are considered true positives and to see how high we need to set our *frequency threshold* for our system to not pick up these alerts. The range for which frequency threshold was tested, were from 0.02 to 0.005.

4.2.3 Experiment 3: False positive reduction

In this experiment we wanted to test how well our system could reduce false positives in an alert log generated from network traffic. We chose to use KDD Cup since it is well documented and it is widely used in academic environments. Having two attack free weeks help in analysing which rules should be accepted or rejected. We have assumed any rules found in the two attack free weeks are false positives while the all others are considered true positives.

5 Results

Results from our experiment described in Section 4.2 were used to evaluate the performance of our false positive reduction system. We will start by looking at results from the honeynet and then proceed to KDD Cup '99. Due to the large amount of data produced by our system at each threshold values we have chosen to only present data from two thresholds, 0.02 and 0.005.

5.1 Initial experiments

Our algorithms use two parameters: *frequency* and *window size*. We performed some initial experiments with varying both window size and frequency threshold. These tests showed that the window size and frequency are somewhat related, e.g. window size: 60 and frequency: 0.01, would produce close to the same results as window size: 30 frequency: 0.005. This was not unexpected and we chose to use a set value for the window size.

We chose to use a set window size of 60, this is the same size as have been used in earlier work [5]. For the purpose of finding alerts related to each other, we find it likely alerts more than 30 seconds apart have not much in common. With the observations discussed in Section 3.3.4 we found it reasonable to use a size of 60. There is no easy way of determining if the chosen value is optimal, as it is highly subjective.

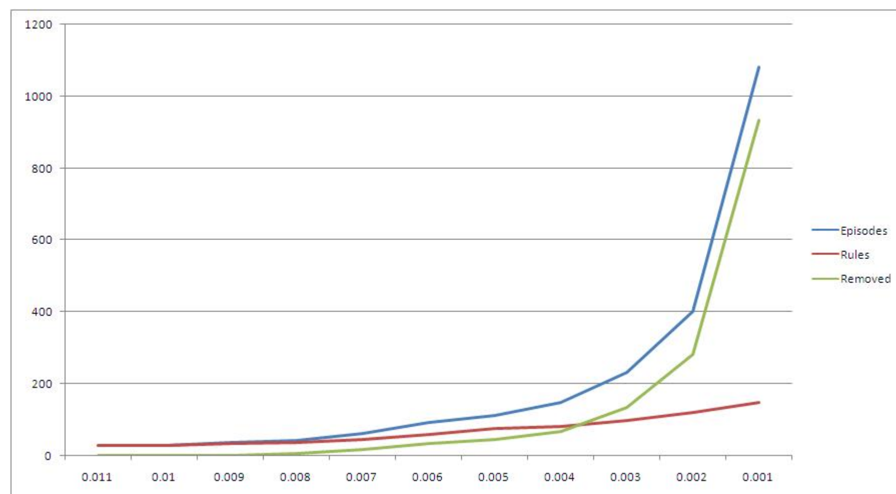


Figure 20: Episodes removed

Figure 20 shows how our episode removal algorithm preforms. The blue line shows how many episodes are found at each threshold while green is the amount of removed episodes. The red line is what we consider "final" episodes, which are used to produce the rules from our system. By studying this graph we observe that with a high threshold no episodes are removed while at lower thresholds the number of removed episodes follows the number of episodes found resulting in few new rules. This demonstrates

our algorithm for removing unwanted episodes is quite effective, leaving only episodes with an impact on the alert log. The number of rules generated does not follow a linear increase with a drop in the threshold, which might be observed from Figure 20 but follows exponential growth. This is severally dampened by the removal of unwanted episodes. The amount of "final" episodes did exceeded 50 episodes even with several thousands of frequent episodes found.

5.2 Experiment 2: Honeynet

The traffic gathered by our honeynet was as expected to mostly from worms and automated bots. Many of the attacks occurred several times a day while others were less frequent, i.e. every other day or once a week. The most common attacks resided from the Code Red, Nimbda and Slammer. Less frequent attacks were brute force attacks on the ftp service. Neither BRO or SNORT picked up on these attacks, they lasted from a few minutes to several hours. Traffic on this network was monitored by both BRO and SNORT intrusion detection system. Our main goal with this network were to check if automated attacks was frequent enough to be picked up.

SNORT raised about 40-55 alerts each day, mostly from Slammer. Each Slammer attack consisted of two alerts: [1:2003:11] and [1:2050:12]. These where found to be quite frequent, and had on average a frequency of 0.0157, with a top frequency of 0.0188 and lowest at 0.0113. Other alerts had a frequency of 0.001 or less, being sporadic pings or port maps. Below we find the single rule found from our honeypot alert log.

```
IF   [1:2003:11]
THEN [1:2050:12]

[1].dst_ip = [2].dst_ip
[1].src_ip = [2].src_ip
[1].src_port = [2].src_port
[1].dst_port = [2].dst_port
[1].ttl = [2].ttl
[1].ip_len = [2].ip_len
[1].dgmlen = [2].dgmlen

[1].dst_port = 1434
[1].dgmlen = 404

[2].dst_port = 1434
[2].dgmlen = 404
```

5.3 Experiment 3: KDD Cup ’99

With the results gathered from our honeypot experiment we decided to show two frequency threshold values, 0.02 and 0.005. First one within the threshold of finding current worm and one outside.

A total of 17063 alerts were generated from the four weeks we used. Each day was

used as a separate input into our mining algorithms. To be able to easily verify which rules from week 4 and 5 modeled false positives we used week 1 and 3. These two weeks do not contain any attacks and any rules found from these two weeks are therefore assumed to be false positives. Rules found in week 4 and 5, which are not part of the set of rules from week 1 and 3 will be assumed to be true positive or ignored. The Table **??** shows the number of alerts generated each day in KDD Cup '99 data set on week 4 and week 5.

The huge increase in alerts in week 5, is due to denial of service attacks. These start on Monday and continue through to Friday, in varying degree. We will take a closer look at this in Section 5.3.2. Week 4 on the other hand does not contain any denial of service attacks and sees a more modest amount of alarms each day compared to week 5.

From week 1 and 3 a total of 35 unique rules where created at a threshold of 0.005.

5.3.1 Week 4

For a threshold value of 0.005 a total of 47 unique rules created for this week, where most of them reside from traffic from the three last days. Some of the rules created from this week is not present in the rules discovered from week 1 and 3. By further analysing these rules, they were found in week 1 and 3 by lowering the threshold just below 0.004. Meaning this traffic is present in those two weeks only less common. The difference in including these or discarding these rules had an impact of about 5% on the amount of filtered alerts. In Figure **??** we have chosen to include these rules. With a threshold value of 0.02 a total of 4 unique rules were created from week 4, all these matched the rules found in week 1 and 3 with the same threshold.

In Figure 24 we show a quick overview of the alerts for each day with remaining alerts after filtering. Tuesday has the lowest reduction in alerts with only 17%. This is the day were inside tcpdump of network traffic is missing, which affects both the number of rules discovered and filtered alerts. Friday is the best day with 85% reduction. The average reduction in alerts for the whole week is 76%.

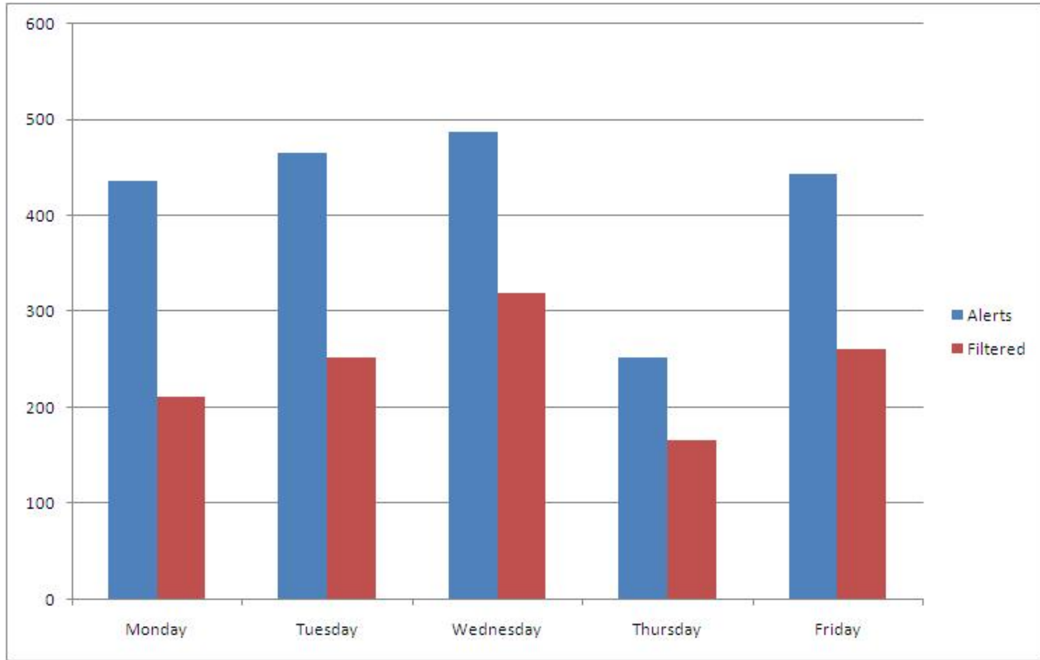
5.3.2 Week 5

This week is quite different from week 4. With denial of service attacks occurring several times has its impact on alarms and the rules discovered. The first thing noticed is the increased time the algorithms use to calculate the rules. The quickest days use 2-3 minutes to suggest rules, while Tuesday uses more than 20 minutes to complete. The completion time is only given as a reference to how time consuming it is, none of the algorithms have been optimized for speed. All times are with an AMD 3700+ CPU.

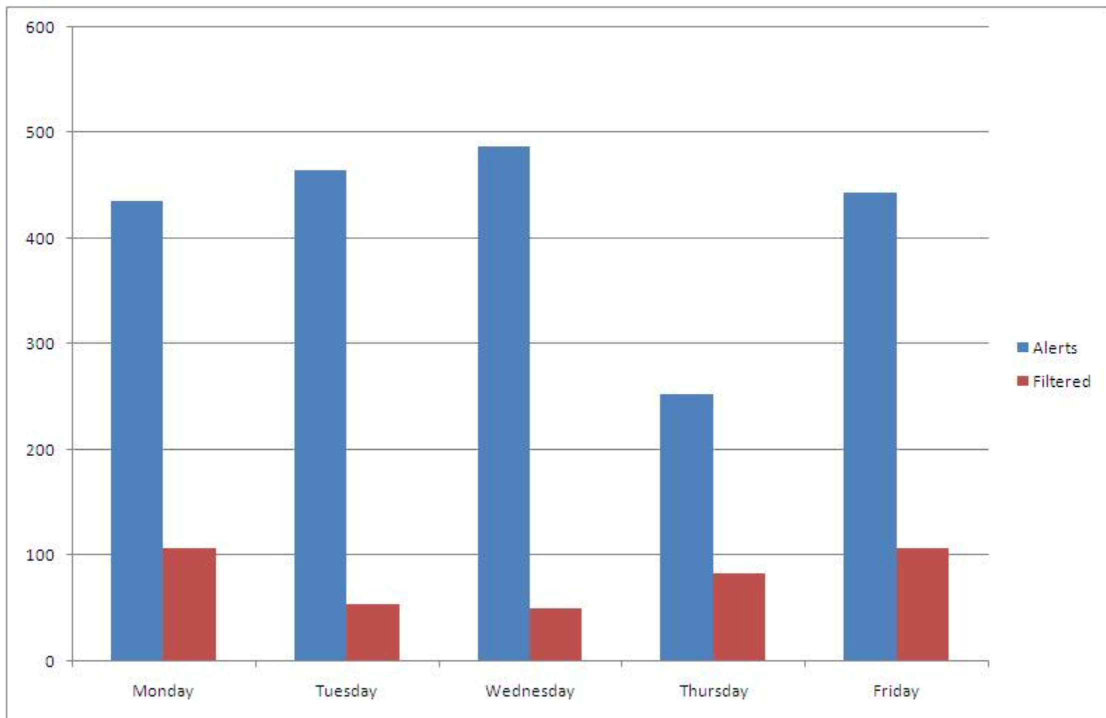
Tuesday has a large denial of service[1] attack which generates several alerts a minute. The attack lasts for 20-25 minutes generating a total 3530 alerts. With our window size of 60 seconds this leaves hundreds of alerts within a single window during a DoS attack. Tuesday had a total of 4197 alerts, this single attack is responsible for 84% of all alerts this day. This day has a large amount of rules, because this day produced an episode with 233 sequential alerts. These rules were quite easy to spot modeling the DoS attack. When the episode rules describing this attack were removed, we were left with 21 rules.

Wednesday is another day with the occurrence of a DoS attack. While the DoS attack on Tuesday lasted about about 20 minutes continuously, on Wednesday there are two short DoS attacks only lasting at max a few minutes. Even with their short time-span

[1] [1:1156:10] WEB-MISC apache directory disclosure attempt, [Classification: Attempted Denial of Service]

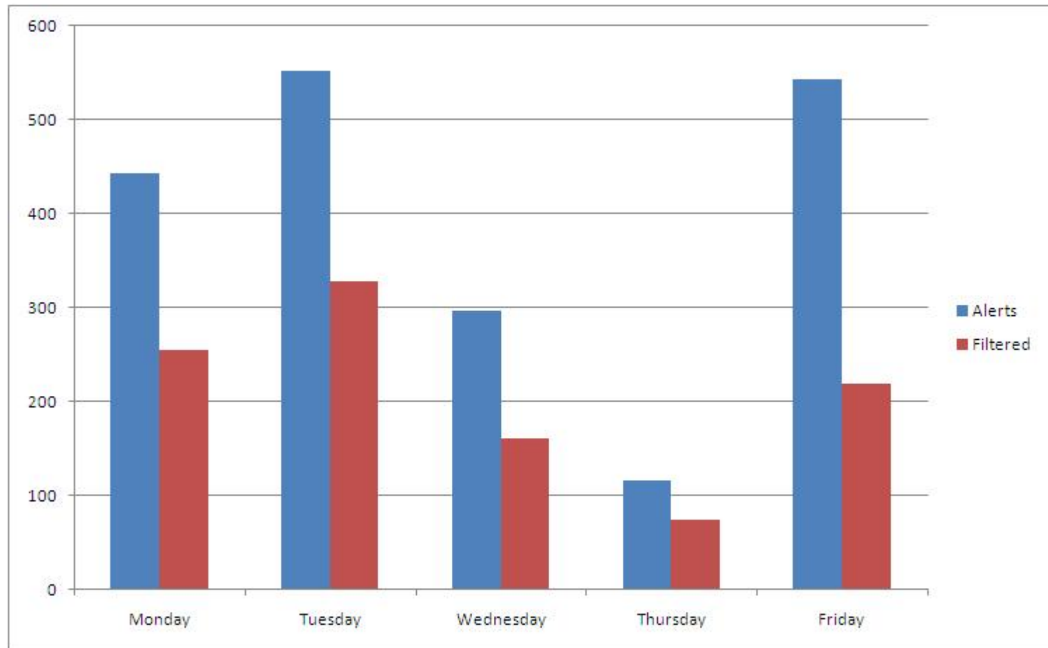


(a) Threshold: 0.02, Window size: 60, average reduction 42%

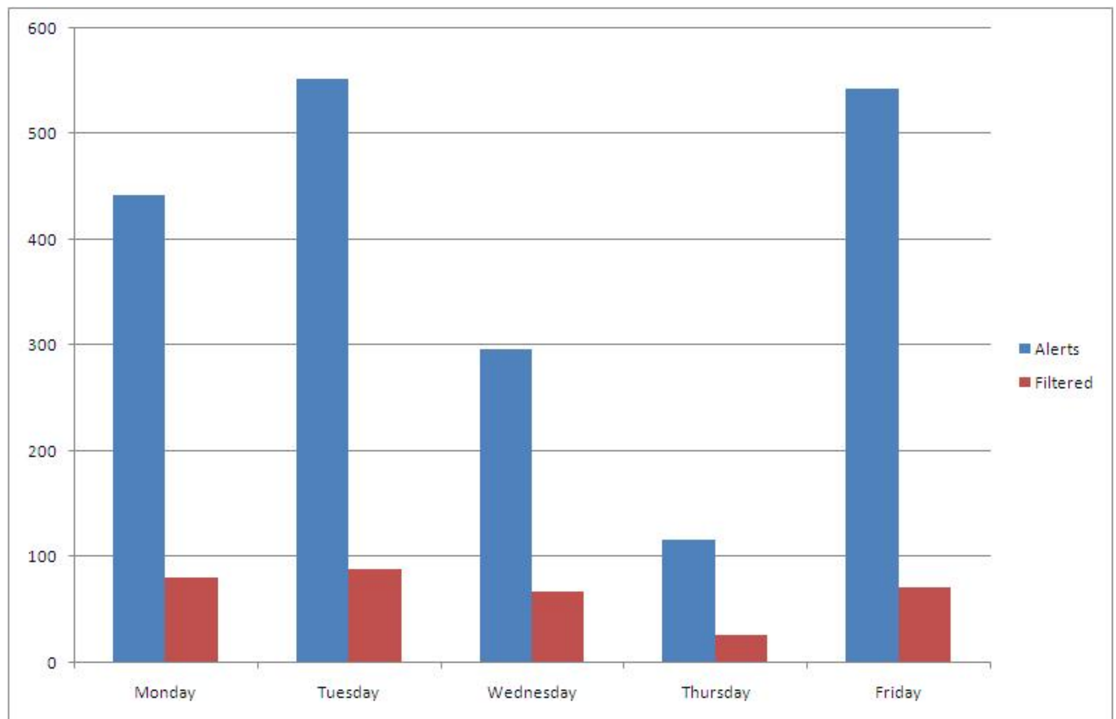


(b) Threshold: 0.005, Window size: 60, average reduction 81%

Figure 21: Alerts in week 1 of KDD Cup '99

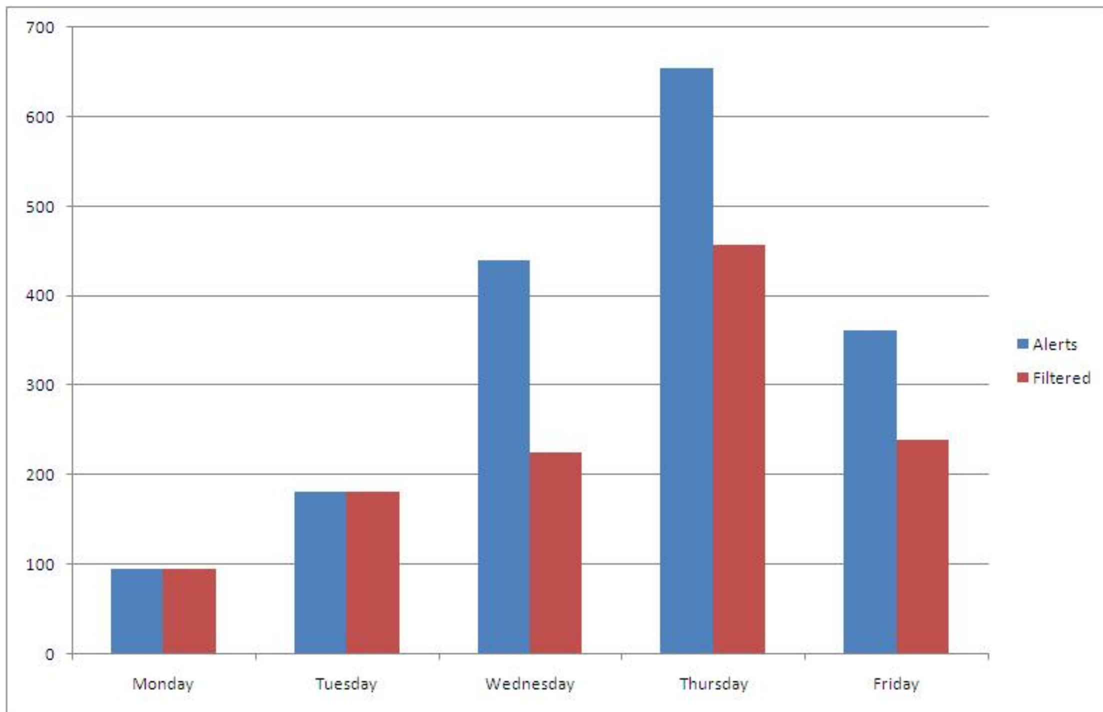


(a) Threshold: 0.02, Window size: 60, average reduction 47%

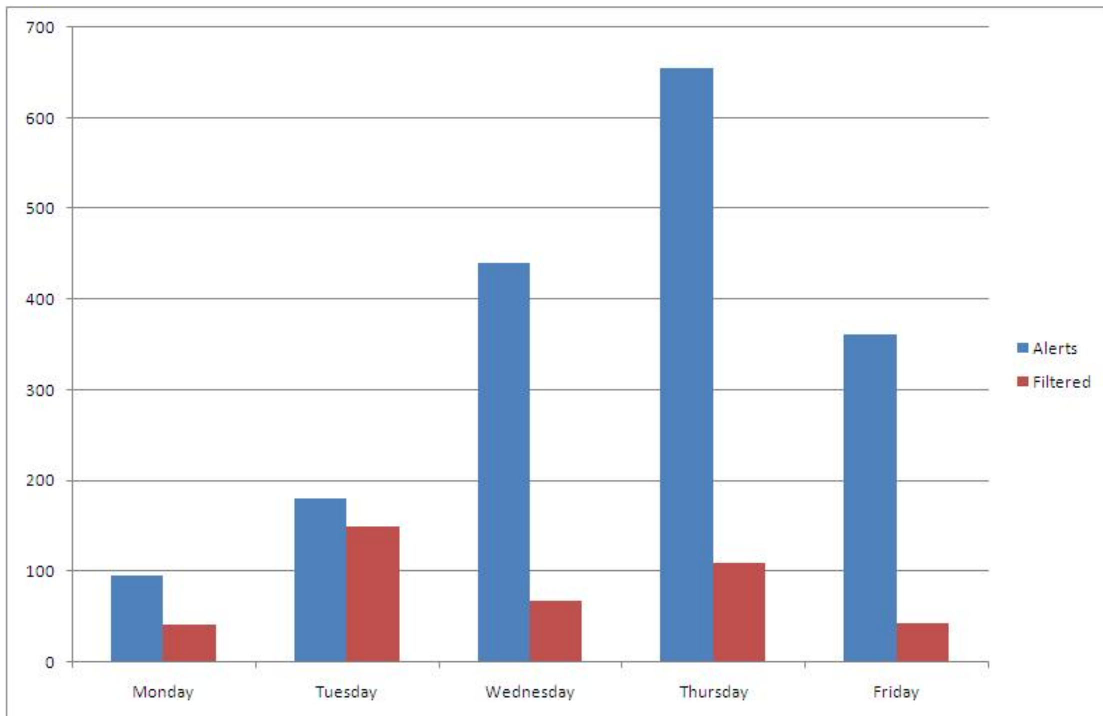


(b) Threshold: 0.005, Window size: 60, average reduction 83%

Figure 22: Alerts in week 3 of KDD Cup '99



(a) Threshold: 0.02, Window size: 60, average reduction 31%



(b) Threshold: 0.005, Window size: 60, average reduction 76%

Figure 23: Alerts in week 4 of KDD Cup '99

they generate 1618 alerts. The reason these alerts did not get recognized as frequent was due to their much shorter time span. With effectively generating hundreds of alerts every second for only a few minutes. When our algorithm collects all the alerts occurring in a single second to create our alert sequence, this results in a lot of alerts getting shifted in and out of our window with a single step of one second. This shows attacks occurring a few times during a day with a short time line, will not be picked up as frequent even if it generates a large amounts of alerts. Friday contains a short DoS attack lasting less than 2 minutes, generating 480 alarms. This one was also not picked up by our algorithms.

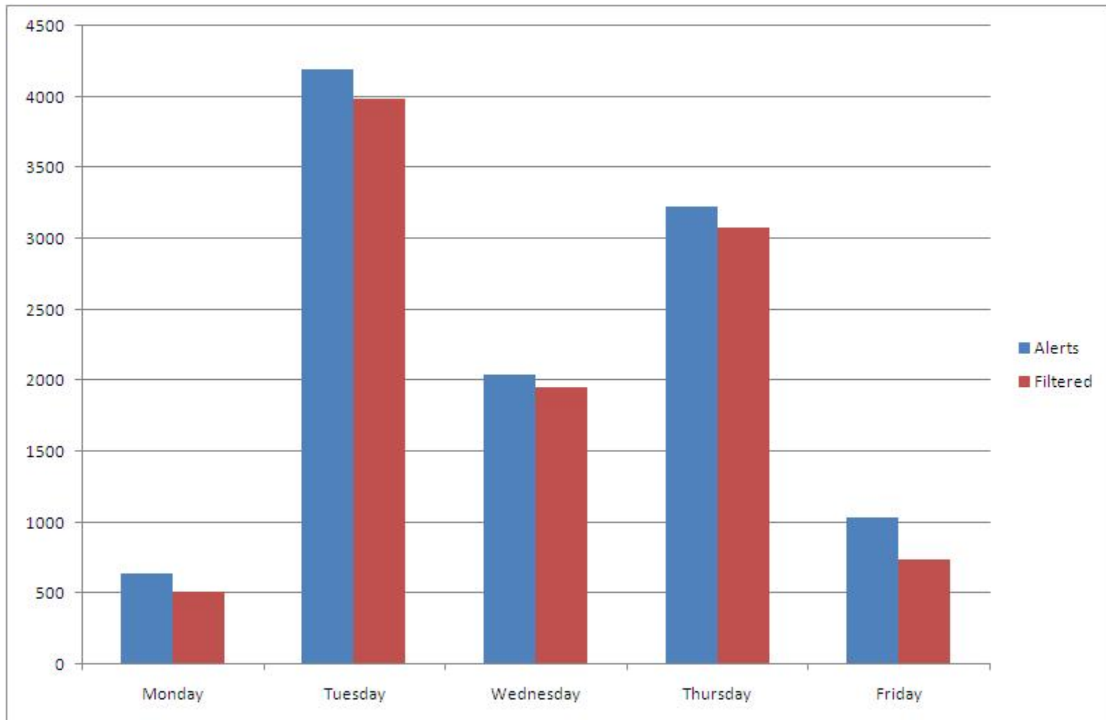
The poor reduction in alerts this week is affected by the high amount of alerts from the DoS attacks occurring. Up to 80% of all alerts generated each day is from the DoS attacks. At a threshold of 0.005 a total of 46 unique rules were created. Some of these rules were found to be true positives, modeling the DoS attack. Even at a threshold of 0.02 a few rules were found to be true positives, these were from the DoS attack on wednesday.

5.4 Discussion

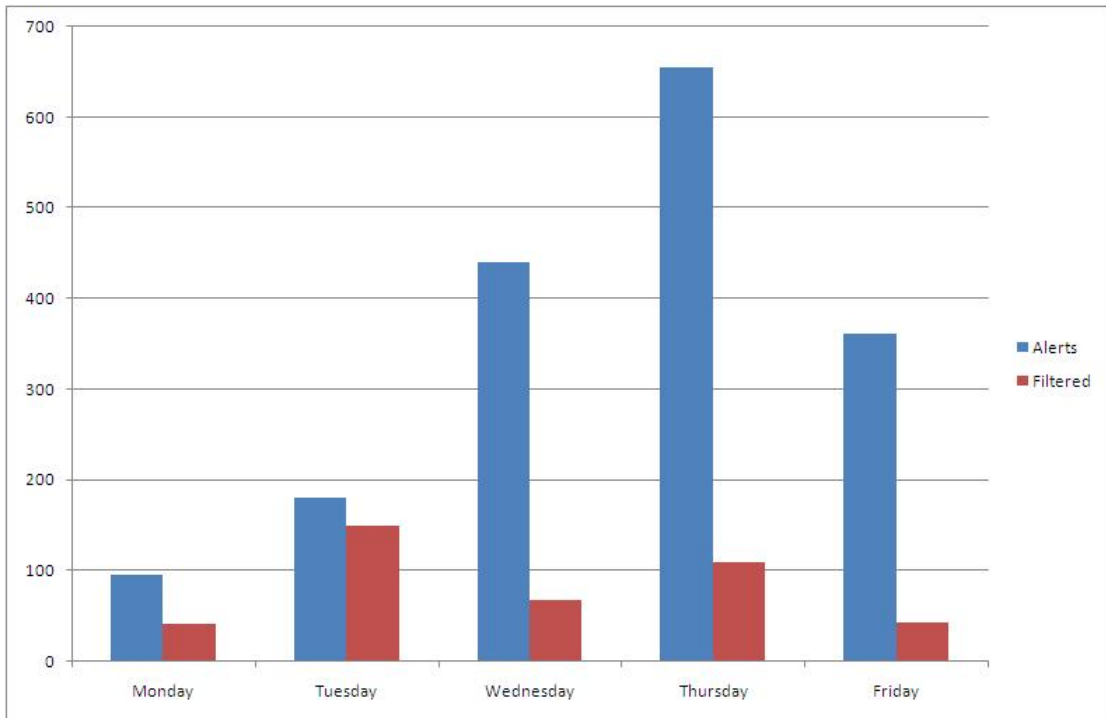
The denial of service attacks in week 5 of KDD Cup were picked up as frequent. This was not unexpected as some of these attacks lasted quite long. The rule generated form this attack became quite large, which was suspicious. Neither BRO nor SNORT picked up the brute force attacks on our FTP server. These attacks lasted anywhere from a few minutes to several hours. Had our IDS generated alerts for these attacks these would most likely be categorised as frequent and rules created. This shows the limitations of our system, if an attack is repeated several times it will be picked up and created a rule for. Therefore the skill of the analyst can determine how effective the system is at reducing false positives. Justification of the analyst being able to classify rules as false positives or not, can be done with the analyst having to be an intrusion detection expert to investigate and analyse alerts to take appropriate actions without our system. The goal of our system is assisting with creating rules for filtering out unwanted false positives. We have used KDD Cup '99 where two weeks are attack free and used these as a baseline for accepting rules. This can create the best case scenario with our setup. Having a data set 100% attack free, it is unrealistic to create from a production network, and an analyst might be more willing to drop rules than accept them.

Even if performance is not the main focus of this thesis we feel some discussion is at hand. Most days with a modest amount of alerts use less than 30 seconds to find rules. One of the biggest factors when it comes to time used is the implementation of the frequent episode algorithms, another big factor is the frequency threshold. The amount of time used is affected by the amount of candidates created trough each pass in the algorithm. The lower the threshold is the more candidates get accepted. The generation of candidates is fast, but testing the presence of each candidate within a window quickly becomes time consuming during discovery of which candidate is frequent. Some days where there were several thousands of candidates could take hours to complete.

A lot of the internet traffic in KDD Cup '99 originates from the same destination. KDD Cup uses a few common addresses to signify that this originates as internet traffic. This might give our results an advantage as more alerts are found to be sent from the same source then they would be in a real network, making a few rules very effective at reducing our alert log. John McHugh's further critiques KDD Cup for its malformed



(a) Threshold: 0.02, Window size: 60, average reduction 8%



(b) Threshold: 0.005, Window size: 60, average reduction 15%

Figure 24: Alerts in week 5 of KDD Cup ’99

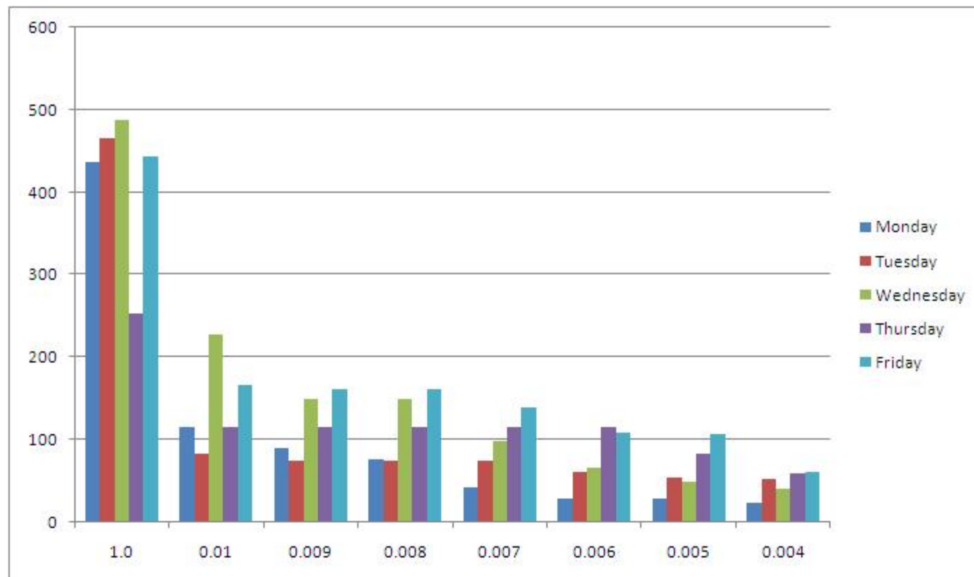


Figure 25: Reduction at different thresholds

internet traffic.

As discussed in Section 3.3.4 the observations of using a window is quite clear when automated attack occur. The alert found on our honeypot originating from worms and bot generally had a very low difference in time between the first alert and the next one. When alerts have a low difference in time, the episode spend more time inside the sliding window in *WINEPI* algorithm making them more likely to be frequent.

There are some risks bound with filtering alerts, to ensure we get the safest possible system three factors should be well understood by the system user. There is no easy way of quantifying the factors beloew making it hard to give an accurate measure.

Analyst The skill of the analyst determine what rules are kept or discard. How safe it is to use a filtering rule depends on the knowledge of the analyst accepting the rules, and the evaluation of the rule.

Attacker Keeping the filtering rules a secret is important to make sure an attacker is not able to design an attack exploiting the filtering rules making the attack go unnoticed.

Environment Many computer environments are highly dynamic, a formerly safe rule might start discarding true positives if the environment has changed. The more static an environment becomes the safer it is to use filtering. In dynamic environment it might be possible to use a life time value on rules, and having each rule rechecked at set intervals.

6 Conclusion

One of the challenges in deploying an IDS in a new environment is getting it set up properly to reduce the false positives. During this thesis we have looked at a method to find patterns in alerts and create rules for filtering false positives out. To find these patterns, frequent episodes (*WINEPI*) have been applied. The rules found are expected to be analysed by a human operator to make the final verdict if the rule holds an alert pattern that is a true positive, false positive or ignored for other reasons. The users of the system should be aware of the important factors affecting the system and its limits discussed in Section 5.4.

To evaluate our approach, a prototype was developed implementing *WINEPI* algorithm along with a rule generator and a filter. SNORT was used with KKD Cup '99 data set to provide an IDS alert log. This data set has seen some criticism regarding its quality, but it was chosen because it has two weeks free of attacks and its widely used in academic environments. Any rules from the attack free weeks would be false positives making it easier to analyse the results. In the real world having a data set modeled from target network which is attack free would be difficult to produce. This might give our system an unfair advantage as it is easy to determine which rules are false positives.

In Section 1.5 we presented two research questions we aimed to answer in this thesis. Here we summarize the answers to these questions, based on the analysis presented in Chapters 5:

1. *Can alerts be effectively be correlated with frequent episodes?*
   Our main concern with frequent episodes was that it is known to produce a large amount of episodes [28, 29, 30]. With a large amount of episodes created into rules, the system user might become swamped by rules and start accepting rules without a thorough analysis. Especially if most of these rules are irrelevant patterns picked up during episode discovery. The latter has been shown to be true due to the way candidates are generated. To solve this problem we developed a way of finding and removing these episodes. What we were left with was a small set of rules that could effectively filter false positives from the alert log. This method proved to be very effective. Even with a large increase in amount of episodes the final rules would not increase by much.

2. *How effective is false positive reduction?*
   Results presented in Chapter 5 show that we got an average 40-80% reduction in alerts. We have shown that our algorithm is able to create a small set of rules which will model a large amount of alerts. These rules can be of great help when deploying intrusion detection systems in new enviroment where getting the correct settings can be hard. There is no guarantee that these rules will model false positives, as we have seen several worm and bot attacks that can be picked up as frequent depending on the frequency threshold. The skill of the analyst to determine which rules to accept and reject has a high influence on how many false positives can be filtered out. The higher the threshold it is set to, the more likely the rules created are false positives,

but reduces the alert log less. A lower threshold will create more rules, giving the analyst more rules to check and verify but has a greater potential to reduce the alert log.

7 Further work

This thesis has studied how episode and association rules can be applied to reduce intrusion detection logs, making it easier for an analyst to find and remove false positives by creating rules. Time and resources have limited our area of focus, by only selecting a few algorithms to work with.

KDD Cup '99 data set consists of many denial of service attacks. These kinds of attacks are not common on most network infrastructures. The data set contains a large amount of attacks which are expected with a data set from a military installation, but less common for other networks. It is possible to filter out attacks, but this has to be done carefully in order not to produce an unrealistic data set. Another big issue discussed earlier it the age of KDD Cup '99 data set, getting close to 10 years old. A lot of new and more sophisticated attacks have emerged. Repeating the experiment on traffic from a production network over a longer time span would be desirable. Other traffic patterns might emerge. There might be different patterns during weekends or off-office hours when maintenance systems start working.

We limited ourselves to the use of *WINEPI* algorithm. There exist quite a few other episode mining techniques, e.g. Fuzzy Episodes, *MINEPI*[9] which might provide different episodes. We used a fairly simple data mining algorithm to create attributes rules from the episodes. An interesting algorithm to use would be association i.e. *ARIROI*[14], an other possibility is using a clustering algorithm instead of association mining on alert attributes to group the matching episodes together.

Bibliography

[1] Hand, D., Mannila, H., & Smyth, P. 2001. *Principles of data mining*. Principles of data mining /David Hand, Heikki Mannila and Padhraic Smyth. Cambridge, Mass. : The MIT Press, c2001. (Adaptive computation and machine learning).

[2] Mannila, H., Toivonen, H., & Verkamo, A. I. 1997. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3), 259–289.

[3] Pietraszek, T. Using adaptive alert classification to reduce false positives in intrusion detection.

[4] Julisch, K. Dealing with false positives in intrusion detection.

[5] Clifton, C. & Gengo, G. 2000. Developing custom intrusion detection filters using data mining. *MILCOM 2000. 21st Century Military Communications Conference Proceedings*, 1, 440–443 vol.1.

[6] Axelsson, S. 1999. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *CCS '99: Proceedings of the 6th ACM conference on Computer and communications security*, 1–7, New York, NY, USA. ACM.

[7] Manganaris, S., Christensen, M., Zerkle, D., & Hermiz, K. 2000. A data mining analysis of rtid alarms. *Comput. Netw.*, 34(4), 571–577.

[8] Leedy, Paul D. Ormrod, J. E. 2001. *Practical Research: Planning and Design.* Prentice-Hall, Inc.

[9] Mannila, H. & Toivonen, H. 1996. Discovering generalized episodes using minimal occurrences. In *Knowledge Discovery and Data Mining*, 146–151.

[10] Bace, R. G. 2000. *Intrusion detection*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA.

[11] Bishop, M. A. 2002. *The Art and Science of Computer Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[12] Kuwatly, I., Sraj, M., Al Masri, Z., & Artail, H. 19-23 July 2004. A dynamic honeypot design for intrusion detection. *Pervasive Services, 2004. ICPS 2004. IEEE/ACS International Conference on*, 95–104.

[13] McHugh, J. 2001. *Intrusion and intrusion detection*. Springer Berlin / Heidelberg.

[14] Inokuchi, A., Washio, T., & Motoda, H. 2000. An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of Data Mining and Knowledge Discovery*, 13–23.

[15] Mannila, H., Toivonen, H., & Verkamo, A. I. 1994. Efficient algorithms for discovering association rules. In *AAAI Workshop on Knowledge Discovery in Databases (KDD-94)*, Fayyad, U. M. & Uthurusamy, R., eds, 181–192, Seattle, Washington. AAAI Press.

[16] Smaha, S. Dec 1988. Haystack: an intrusion detection system. *Aerospace Computer Security Applications Conference, 1988., Fourth*, 37–44.

[17] Snort. http://snort.org/ (Last visited March 2008).

[18] Paxson, V. 1999. Bro: a system for detecting network intruders in real-time. *Comput. Netw.*, 31(23-24), 2435–2463.

[19] Ilgun, K. 24-26 May 1993. Ustat: a real-time intrusion detection system for unix. *Research in Security and Privacy, 1993. Proceedings., 1993 IEEE Computer Society Symposium on*, 16–28.

[20] Kumar, S. *Classification and detection of computer intrusions*. PhD thesis, West Lafayette, IN, USA, 1995.

[21] Tanenbaum, A. S. 1996. *Computer networks*. Englewood Cliffs: Prentice-Hall, |c1996, 3rd ed., international edition.

[22] Georges, J. & Milley, A. H. 2000. Kdd'99 competition: knowledge discovery contest. *SIGKDD Explor. Newsl.*, 1(2), 79–84.

[23] Kohavi, R., Brodley, C. E., Frasca, B., Mason, L., & Zheng, Z. 2000. Kdd-cup 2000 organizers' report: peeling the onion. *SIGKDD Explor. Newsl.*, 2(2), 86–93.

[24] Cheng, J., Hatzis, C., Hayashi, H., Krogel, M.-A., Morishita, S., Page, D., & Sese, J. 2002. Kdd cup 2001 report. *SIGKDD Explor. Newsl.*, 3(2), 47–64.

[25] Lau, S. 2004. The spinning cube of potential doom. *Commun. ACM*, 47(6), 25–26.

[26] Lee, W. A data mining framework for constructing features and models for intrusion detection systems.

[27] Lee, W., Stolfo, S., & Mok, K. 1999. A data mining framework for building intrusion detection models. *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on*, 120–132.

[28] Qin, M. & Hwang, K. 2004. Frequent episode rules for internet anomaly detection. In *NCA '04: Proceedings of the Network Computing and Applications, Third IEEE International Symposium*, 161–168, Washington, DC, USA. IEEE Computer Society.

[29] Julisch, K. 2003. Using root cause analysis to handle intrusion detection alarms.

[30] Klemettinen, M. 1999. A knowledge discovery methodology for telecommunication network alarm databases.

[31] Lee, W. & Stolfo, S. J. 1998. Data mining approaches for intrusion detection. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium, 1998*, 6–6, Berkeley, CA, USA. USENIX Association.

[32] Mell, P., Hu, V., Lipmann, R., Haines, J., & Zissman, M. An overview of issues in testing intrusion detection systems.

[33] Bishop, M.; Bhumiratana, B. C. R. L. K. 14-16 June 2004. How to sanitize data? *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2004. WET ICE 2004. 13th IEEE International Workshops on*, 217–222.

[34] Durst, R., Champion, T., Witten, B., Miller, E., & Spagnuolo, L. 1999. Testing and evaluating computer intrusion detection systems. *Commun. ACM*, 42(7), 53–61.

[35] Lippmann, R., Fried, D., Graf, I., Haines, J., Kendall, K., McClung, D., Weber, D., Webster, S., Wyschogrod, D., Cunningham, R., & Zissman, M. 2000. Evaluating intrusion detection systems: the 1998 darpa off-line intrusion detection evaluation. *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, 2, 12–26 vol.2.

[36] Lippmann, R., Haines, J. W., Fried, D. J., Korba, J., & Das, K. 2000. The 1999 darpa off-line intrusion detection evaluation. *Comput. Netw.*, 34(4), 579–595.

[37] 2000. Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory. *ACM Trans. Inf. Syst. Secur.*, 3(4), 262–294.

[38] Oberheide, J. & Karir, M. Honeyd detection via packet fragmentation.

[39] Send-safe honeypot hunter. http://send-safe.com/honeypot-hunter.html (Last visited February 2008).

[40] Krawetz, N. Jan.-Feb. 2004. Anti-honeypot technology. *Security and Privacy, IEEE*, 2(1), 76–79.

[41] Provos, N. Honeyd virtual honeypot. http://www.honeyd.org/ (Last visited March 2008).

[42] Inc., N. Netbait. http://netbaitinc.com/ (Last visited March 2008).

[43] Honeynet project. www.honeynet.org (Last visited February 2008).

A Source Code

Here we list samples of the source code used in the implementation of our system. The prototype is written in C++ and we give a short description of the functions listed.

A.1 Creating new serial candidates

Implementation of Algorithm 3, described in Section 3.3.2.

```
Episodes serial_candiates(Episodes f)
{
  Episodes  c;
  Episode   a;
  Episode   b;
  int k = 0;
  int l = 0;
  int current_block_start = 0;
  bool flag = true;



  //if there are no frequent episodes there
  //won't be any new candidates either.
  if( f.size() > 0 )
    l = f[0].size();
  else
    return c;



  for(int i=0; i<f.size(); ++i)
  {
    current_block_start = k + 1;
    for(int j=f[i].block_start; (j < f.size()) && f[j].block_start == f
       ↪[i].block_start; ++j )
    {
      /* f[i] and f[j] have l-1 event types in common, build a
         potential candidate a as their combination
      */
      for(int x=0; x<f[i].events.size(); ++x)
      {
        a.events.push_back( f[i].events[x] );
      }
      std::vector<Event> tmp;
      tmp.push_back( f[j].events.back().back() );
      a.events.push_back( tmp );


      /* Build and test subepisodes b that do not contain a[y] */
      for(int y=0; y < l; ++y)
      {
        b = a.sub_episode(y);

        //if b is not in f then continue with the next a at line 6
```

```
        if( is_episode_in_collection(f, b) == false )
          flag = false;
      }

      /* All subepisodes are in f, store a as candidate */
      if( flag )
      {
        k++;
        c.push_back( a );
        (c.end()-1)->block_start = current_block_start;
      }

      flag = true;
      a.clear();
      b.clear();
    }
  }

  return c;
}
```

A.2 Discovering frequent episodes

Implementation of Algorithm 4, described in Section 3.3.3

```
Episodes find_frequent_episodes(EventSequence s, Episodes c, int win,
    ↪float min_fr)
{
  std::vector<Event> window;

  /* Initialization */
  for(int i=0; i<c.size(); ++i)
  {
    for(int j=0; j<s.event_types.size(); ++j)
      s.event_types[j].count = 0;

    c[i].event_count = 0;
    c[i].freq_count = 0;
    c[i].is_in_window = false;
  }


  /* Recognition */
  for(int start = (s.start_time-win+1); start <= (s.end_time+1); ++
      ↪start)
  {
    /* Bring new events to the window */
    std::vector<Event> local_events = s.get_events_at( start+win-1 );

    //copy all the event
    std::copy( local_events.begin(), local_events.end(), std::inserter(
        ↪window, window.begin()) );

    for(int i=0; i<local_events.size(); ++i)
    {
      Event &et = s.event_types[ local_events[i].id ];
      et.count += 1;
```

```
      //returns all episodes that contains Event (et), and a specified
         ↪number of that event (count)
      std::vector<Episode*> e = contains(c, et, et.count);
      for(int i=0; i<e.size(); ++i)
      {
        e[i]->event_count += et.count;

        if( e[i]->event_count == e[i]->size() )
        {
          //all the events for the episode is inside the window

          //test partial order of the window and that it conforms with
             ↪the episode
          if( e[i]->test( EventSequence(window)) )
          {
            e[i]->in_window = start;
            e[i]->is_in_window = true;
          }
        }
      }
    }

    /* Drop out old events from the window */
    local_events = s.get_events_at( start-1 );
    if(local_events.empty() == false && window.empty() == false)
      for(int i=0; i<local_events.size(); ++i)
        window.pop_back();

    for(int i=0; i<local_events.size(); ++i)
    {
      Event &et = s.event_types[ local_events[i].id ];
      std::vector<Episode*> e = contains(c, et, et.count);
      for(int j=0; j<e.size(); ++j)
      {
        if( e[j]->is_in_window && e[j]->event_count == e[j]->size() )
        {
          e[j]->is_in_window = false;
          e[j]->freq_count = e[j]->freq_count + (start - e[j]->
             ↪in_window);
        }

        e[j]->event_count -= s.event_types[ local_events[i].id ].count;
      }

      s.event_types[ local_events[i].id ].count--;
    }
  }

  /* Output */
  Episodes output;
  for(int i=0; i<c.size(); ++i)
  {
    c[i].fr = (float)c[i].freq_count/(s.end_time-s.start_time+win-1);
    //std::cout << c[i].print() << std::endl;

    if( c[i].fr >= min_fr )
    {
      output.push_back(c[i]);
```

```
    }
  }

  return output;
}
```

A.3 Remove unwanted episodes/Filter alert sequence

Implementation Algorithm 5 and the filter, described in Section 3.3.5 and 3.5. This algorithm finds all "final" episodes, those episodes not used to remove any alerts from an alert sequence are removed. The actual implementation of the filter is very similar to this algorithm an only changes a few lines. Instead of increasing e.hit_count and e.sub_hit, the actual event are removed from the window. The difference between is listed below, this is what needs change in the "bool is_episode_present(...)" function

```
    if( serial_event == e.events.size() )
    {
      e.hit_count++;
      e.increase_sub_episode_hit();
      return true;
    }

    if( serial_event == e.events.size() )
    {
      for(int i=0; i<index.size(); ++i)
      {
        window.erase( window.begin()+index[i] );   //remove matched
            ↪alerts from window
      }
    }


void remove_unwanted_episodes(Episodes& frequent_episodes, int win,
    ↪EventSequence event_stream)
{
  std::vector< Event > window;

  int start_window = 0;
  int end_window = win;


  //init
  for(int i=0; i<end_window; ++i)
  {
    std::vector<Event> local = event_stream.get_events_at(i);

    for(int j=0; j<local.size(); ++j)
      window.push_back( Event(local[j]) );
  }


  for(; end_window < event_stream.end_time; ++end_window, ++
      ↪start_window)
  {
    //remove events < start_window
    while( window.empty() == false && start_window > window[0].time )
    {
```

```
      window.erase( window.begin() );
    }

    //add new events at, end_window
    std::vector<Event> local = event_stream.get_events_at(end_window);

    for(int j=0; j<local.size(); ++j)
      window.push_back( Event(local[j]) );


    //test if the frequent episode is present in the window
    for(int i=0; i<frequent_episodes.size(); ++i)
    {
      if( is_episode_present(frequent_episodes[i], window) )
        break;
    }
  }

  //remove unwanted episodes
  for(int i=0; i<frequent_episodes.size(); ++i)
  {
    if(frequent_episodes[i].hit_count == 0)
      frequent_episodes.erase( frequent_episodes.begin()+i );
  }
}


bool is_episode_present(Episode &e, std::vector<Event> &window)
{
  int serial_event = 0;
  std::vector<int> index;

  std::vector<int> event_index;

  for(int i=0; i<window.size();)
  {
    for(int j=i;j<window.size();++j)
    {
      if( e.events[serial_event][0].id == window[j].id )
        event_index.push_back(j);
    }

    if(event_index.size() > 1)
    {
      std::vector<int> best_match;
      for(int i=0; i<event_index.size(); ++i)
      {
        best_match.push_back( penalty_value(e.events[serial_event][0],
            ↪window[ event_index[i] ]));
      }

      std::sort(best_match.begin(), best_match.end());

      index.push_back( *(best_match.end()-1) );
      serial_event++;

      i = (best_match.end()-1);
    }
```

```
    else if(event_index.size() == 1)
    {
      index.push_back(event_index[0]);
      serial_event++;

      i = event_index[0];
    }

    if( serial_event == e.events.size() )
    {
      e.hit_count++;
      e.increase_sub_episode_hit();
      return true;
    }

}
```

B Rules, Wednesday Week 4

Below we give a listing of the rules found from Wednesday in week 4 of KDD Cup '99 with a threshold value of 0.005 and a window size of 60.

```
Rule1:

IF    [1:1013:11]
THEN  [1:1012:12]  conf(0.353) freq(0.006)
      [1:1288:10]

IF    [1:1013:11]
      [1:1012:12]
THEN  [1:1288:10]  conf(1.0) freq(0.006)

[1].src = [2].src = [3].src
[1].dst = [2].dst = [3].dst
[1].src_port = [2].src_port = [3].src_port
[1].dst_port = [2].dst_port = [3].dst_port
[1].ttl = [2].ttl = [3].ttl
[1].dgmlen = [2].dgmlen = [3].dgmlen

[1].dst_port = 80
[2].dst_port = 80
[3].dst_port = 80
[1].ttl = 64
[2].ttl = 64
[3].ttl = 64
[1].src = 172.16.115.87
[2].src = 172.16.115.87
[3].src = 172.16.115.87
[1].dst = 209.61.100.129
[2].dst = 209.61.100.129
[3].dst = 209.61.100.129



Rule2:

IF    [1:1149:13]
THEN  [1:1149:13]  conf(0.53) freq(0.007)

[1].src = [2].src
```

```
[1].dst = [2].dst
[1].dst_port = [2].dst_port
[1].ttl = [2].ttl

[1].dst_port = 80
[2].dst_port = 80
[1].ttl = 64
[2].ttl = 64


Rule3:

IF    [1:1013:11]
THEN  [1:1288:10]  conf(1.0) freq(0.017)

[1].src = [2].src
[1].dst = [2].dst
[1].dst_port = [2].dst_port
[1].src_port = [2].src_port
[1].dgmlen = [2].dgmlen
[1].ttl = [2].ttl

E[1].dst_port = 80
E[2].dst_port = 80
E[1].ttl = 64
E[2].ttl = 64



Rule4:

IF    [1:1012:12]
THEN  [1:1288:10]  conf(1.0) freq(0.006)

[1].src = [2].src
[1].dst = [2].dst
[1].src_port = [2].src_port
[1].dst_port = [2].dst_port
[1].ttl = [2].ttl
[1].dgmlen = [2].dgmlen

[1].dst_port = 80
[2].dst_port = 80
[1].ttl = 64
[2].ttl = 64
[1].src = 172.16.115.87
```

```
[2].src = 172.16.115.87
[1].dst = 209.61.100.129
[2].dst = 209.61.100.129



Rule5:

IF  [1:1013:11]
THEN [1:1012:12]  conf(0.353) freq(0.006)

[1].src = [2].src
[1].dst = [2].dst
[1].src_port = [2].src_port
[1].dst_port = [2].dst_port
[1].ttl = [2].ttl
[1].dgmlen = [2].dgmlen

[1].dst_port = 80
[2].dst_port = 80
[1].ttl = 64
[2].ttl = 64
[1].src = 172.16.115.87
[2].src = 172.16.115.87
[1].dst = 209.61.100.129
[2].dst = 209.61.100.129
```

C Installation

C.1 Bro IDS

Debian was our choice as OS. A standard install with bar minimum is all that is needed. It needs two network interfaces as it will act as a bridge connecting our honeypots to HiG's IDS network, analyzing traffic passing through it. All network addresses are based on **??** There are no pre-compiled packages provided for Bro IDS, so we need to download and build the source our self. In order to do so we need some tools to help us. The following commands will install both libraries and the tools needed by Bro to build it.

Dependancies:

```
# apt-get install libncurses-dev
# apt-get install tcpdump
# apt-get install libpcap0.7-dev
# apt-get install autoconf
# apt-get install automake
# apt-get install build-essentials
# apt-get install flex
# apt-get install bison
```

The three first packages is needed by Bro IDS while the rest is needed to compile the source code. When the above packages have been installed the next step is to download the source, it can be found at *ftp://bro-ids.org/*. For our installation we choose to use *bro-1.2-stable.tar.gz*. When we have downloaded Bro we need to extract, build and finally, install it.

Extract:

```
# tar -xzf bro-1.2-stable.tar.gz
```

Build:

```
# ./configure --diable-broccoli --prefix=/usr/local/bro
# make
```

Install:

```
# make install
# make install-brolite
```

We have no need for broccoli [1] so we disable it. Broccoli enables several Bro clients to communicate together. After bro has been installed it will reside in */usr/local/bro/* on the computer. Before we start running Bro, we need to make a few adjustments in the configuration files.

Setup:

[1] Broccoli: BRO Client COmmunications LIbrary.

```
# nano /usr/local/bro/site/local.site.bro, fill in network details.
# nano /usr/local/bro/site/<computer_name>.bro, uncomment line "# @local.brolite-sig" to enab
```

Now Bro is ready to run. But before we can start using it we need to set it up as a bridge connecting our honeypots through the Bro IDS system.

```
apt-get install bridge-utils
ifconfig eth0 0.0.0.0 arp -promisc up
ifconfig eth1 0.0.0.0 arp -promisc up
brctl addbr br0
brctl addif br0 eth0
brctl addif br0 eth1
route add default gw 128.39.44.1
```

Run:

```
# /usr/local/bro/etc/bro.rc start
```

D Installing HoneyWall

Honeywall installation is fully automated, taking about 5-10 minutes to complete. It will format and partition the drive by it is own. When it is done installing some final configuration has to be done. Once the installation is done login is done with a preset user and password:

```
# login: roo
# password: honey
```

It is crucial to change name and password after login since these are set default values. To get root access, type

```
 # su -
```

with "honey" as password. This of course needs change too. Now with root access it is possible to do the final configuration, this is started with "menu". It starts a basic gui which helps with configuring. Most crucial is network information, other settings often come with default values that can be left alone. Another important thing during configuration is which two network interface controllers will be used in bridge mode . A problem is to locate these on the back side of the computer. This can be done with only connecting one network interface to internet and trying use e.g. ping to try and reach a host. Testing all ports in order we are able to learn what each port is called in linux (i.e eth0, eth1 and eth2).