Anomaly Detection in Computer Networks
Using Hierarchically Organized Teams of Learning Automata

Vegard Haugland, Marius Kjølleberg and Svein-Erik Larsen

Supervisors
Stein Bergsmark and Ole-Christoffer Granmo

This Master’s Thesis is carried out as a part of the education at the University of Agder and is therefore approved as part of this education.

University of Agder
Faculty of Engineering and Science
Department of ICT, 2011
Abstract

With the increasing number of computer systems connected to the Internet, security becomes a critical issue. To combat this problem, several attack detection methods have emerged in the past years, such as the rule based Intrusion Detection System (IDS) *Snort* - or anomaly based alternatives that are able to detect novel attacks without any prior knowledge about them.

Most current anomaly based IDS require labeled attacks or extensively filtered training data, such that certain attack types, which generate large amounts of noise in terms of false positives, are effectively removed.

This thesis describes a novel anomaly based scheme for detecting attacks, using frequent itemset mining, without performing extensive filtering of the input data. In brief, the scheme, which is named the Grimstad Data Classifier (*GRIDAC*), uses teams of hierarchically organized Learning Automata to generate a rule tree with a set of linked nodes – where the granularity of each node increases along with the current level in the tree.

In turn, *GRIDAC* was implemented as an anomaly based IDS called *Inspectobot*, and evaluated using the 1999 DARPA IDS Evaluation Sets. At best, the prototype was able to detect 51 out of 62 attacks in the 1999 DARPA IDS Evaluation Sets with 56 false alarms, giving a detection rate of 82 %, after training on one week of attack-free traffic, and classifying another full week of data containing attacks.

The empirical results are quite conclusive, demonstrating that the prototype shows an excellent ability to mine frequent itemsets from network packets, such that normal behavior can be modeled. With an average detection rate of 73 % of all attacks in the DARPA set, and a fairly low amount of false positives, it is also shown that *Inspectobot* can be used for IDS purposes.

In its current state, *Inspectobot* requires a high processing capacity to perform the rule matching. When compared to the popular IDS *Snort*, it is currently not as useful outside of a testbed environment. Nonetheless, the scheme has the potential of serving as a complementary anomaly based IDS alongside *Snort* for detecting novel attacks, given a more optimized implementation.
Preface

This Master’s Thesis was submitted in partial fulfillment of the requirements for the degree Master of Science in Information and Communication Technology (ICT) at the University of Agder – where the workload is set to a total of 30 ECTS credits.

The main idea behind the project, using hierarchically organized teams of learning automata to detect anomalies in computer networks, was suggested by the team members along with associate professor Ole-Christoffer Granmo. The team members are all attending the security profile in the Master’s Degree program in ICT at the Faculty of Engineering and Science, University of Agder. Also, the project has been carried out under the supervision of associate professor Ole-Christoffer Granmo and project manager Stein Bergsmark.

Work on the project started in late October. Early on, a platform for development, report writing and collaboration was established. Revision control for all written code, text and data was provided by Git*. In addition, a minimalistic project management system called Trac† was used. Trac features a comprehensive wiki for collaborative documentation, an issue tracking system for software development projects, in addition to tools for project management. A web based interface to the revision control system Git is also included, making it easier to get a detailed view of all the current data in the repository.

For internal communication between the members, a private channel was established using Internet Relay Chat (IRC), TaskJuggler‡ was used to create GANTT charts, and \LaTeX was used for document formatting and preparation.

The team would like to thank project manager Stein Bergsmark, for his insight and advice on teamwork and development processes - and associate professor Ole-Christoffer Granmo for the project idea and his expertise in the machine learning domain. Both have been great in assisting with report writing and providing helpful comments. Additional thanks goes to David Cowen, CISSP, and Professor Jose J. Cabeza Gonzalez, for their valuable input.

Grimstad, 2011

Vegard Haugland, Marius Kjølleberg and Svein-Erik Larsen

# Contents

<table>
<thead>
<tr>
<th>Abstract</th>
<th>ii</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>iii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Background and Motivation ........................................... 2
1.2 Problem and Research Questions .................................... 4
1.3 Literature Review ..................................................... 6
   1.3.1 Packet Header Inspection ................................. 6
   1.3.2 PHAD and NETAD ........................................... 7
   1.3.3 Unsupervised Anomaly Detection in Network Intrusion Detection Using Clusters ...................... 9
   1.3.4 MINDS - Minnesota Intrusion Detection System ............. 10
   1.3.5 An Assessment of the DARPA IDS Evaluation Dataset using Snort ........................................... 12
1.4 Method ....................................................................... 12
   1.4.1 Solution Approach ........................................... 12
   1.4.2 Software Development Approach ............................ 16
   1.4.3 Choice of Programming Language ........................... 17
   1.4.4 Quality Assurance ........................................... 18
1.5 Key Assumptions and Limitations .................................... 19
1.6 Contribution to Knowledge .......................................... 20
1.7 Thesis Outline ......................................................... 20

## 2 Machine Learning and Applications

2.1 Reinforced Learning .................................................. 22
2.2 Learning Automata ..................................................... 23
2.3 Tsetlin Automaton ...................................................... 24
2.4 Patterns in Network Traffic .......................................... 25

## 3 Solution

3.1 Requirements .......................................................... 26
3.2 Work Package Overview ............................................. 29
3.3 Action Selection ....................................................... 30
3.4 Values of $r$ With Respect to $x$ .................................. 31
3.5 Multiple Rules and Hierarchical Organization .................... 39
3.6 GRIDAC as an A-NIDS ............................................... 42
## Testing and Validation

### 4.1 Test Programme

### 4.2 GRIDAC Parameter Tuning

### 4.3 Classification Evaluation with Artificial Data

### 4.4 IDS Evaluation with Artificial Data

### 4.5 Classification Evaluation with Network Packets

### 4.6 IDS Evaluation with Network Packets

## Discussion

### 5.1 Frequent Pattern Mining in Network Packets

### 5.2 Potential as an Intrusion Detection System

### 5.3 Possible replacement for Snort?

### 5.4 Remarks concerning the 1999 DARPA IDS Evaluation Sets

## Conclusion

### 6.1 Empirical Results

### 6.2 Conclusions and Implications

### 6.3 Future Work

Bibliography

Appendix A - Work Package Example

Appendix B - Test Case Example

Appendix C - GANTT Chart
List of Figures

1.1 Reported Security Incidents by Year .................................. 2
1.2 System Archetype of Traditional Signature Based IDS ....... 3
1.3 NETAD Attribute Model as a Boolean Expression .......... 8
1.4 ROC curve of fpMAFIA ........................................... 10
1.5 MINDS System .................................................... 11
1.6 Selection of split criteria ........................................ 13
1.7 Hierarchical structuring .......................................... 15
1.8 System development using the Prototyping method ......... 17

2.1 Hungry Mouse in a T-Shaped Maze ................................. 22
2.2 Relation between actions and rewards, with regards to the environment ................................................. 23
2.3 Two-action Tsetlin Automaton with 3 states per action .... 24
2.4 IPv4 Packet Header Structure .................................... 25

3.1 Tsetlin Automaton with n states per action and action probabilities ......................................................... 31
3.2 Markov Chain ......................................................... 33
3.3 Filter Object Selection ............................................ 39
3.4 Hierarchical Organization of Abstract Objects .............. 40
3.5 Inspectobot’s architectural design ................................. 42
3.6 Example Rule ........................................................ 47
3.7 Graphical representation of a rule hierarchy ................... 48
3.8 E-R diagram of Inspectobot’s Database Structure ........... 49
3.9 Screenshot of Inspectobot in Action .............................. 50
3.10 Screenshot of Inspectobot in Action 2 ......................... 51

4.1 Number of required iterations with regards to the number of states per action ......................................... 55
4.2 Template objects present in DS 1 ................................. 57
4.3 Rule classifying 75% of the objects in DS 1 ................... 57
4.4 Examples of objects present in DS 3 and DS 4 ............... 57
4.5 Various template objects used for training Inspectobot ... 59
4.6 Various anomalous template objects used for testing Inspectobot ......................................................... 59
4.7 Anomaly score distribution for an artificial dataset ........ 60
4.8 Impact of different split criteria in a given rule ............ 62
4.9 Example rule generated with a classification target of 50% . 62
## List of Tables

1.1 An example rule generated from the abstract dataset shown in Figure 1.6 ........................................ 14
1.2 Another example rule generated from the abstract dataset shown in Figure 1.6 ................................. 14
1.3 Example rule that classifies the objects shown in Table 1.1 and Table 1.2 ........................................ 14
3.1 Work Package Overview ........................................ 29
3.2 Values of $r$ when $0 < x < 1$ ................................ 32
3.3 Values of $r$ using Equation 3.4 .............................. 33
3.4 Action probabilities when $r < 1$ ............................. 35
3.5 Action probabilities when $r \geq 1$ ........................... 36
3.6 Parameters Related to Dataset Parsing .................... 46
3.7 Parameters Related to Dataset Training ................... 47
3.8 Rule Tree Symbol Explanation .............................. 48
4.1 Test Programme .................................................. 53
4.2 Test Case 5 - Subtests ........................................... 54
4.3 Parameter Values used in Test Programme .............. 54
4.4 Dataset Overview ............................................... 56
4.5 Results from testing the classification aspect of Inspectobot ..................................................... 58
4.6 Rule examples and some of the attacks they can detect ... 66
4.7 Results from IDS evaluation using the DARPA set ...... 70
4.8 Undetected DARPA Attack Types .......................... 70
5.1 Pros and Cons with Inspectobot ............................. 80
TC5.1 Test Setup ................................................... 97
TC5.2 Table explanation ........................................... 98
TC5.3 Week 1 - Outside - Node Limit 1% ...................... 99
TC5.4 Week 1 - Outside - Node Limit 5% ...................... 99
TC5.5 Week 1 - Outside - Node Limit 10% .................... 99
TC5.6 Week 3 - Outside - Node Limit 1% ..................... 100
TC5.7 Week 3 - Outside - Node Limit 1% ..................... 100
TC5.8 Week 3 - Outside - Node Limit 10% .................... 100
TC5.9 Week 1+3 - Outside - Node Limit 1% .................. 101
TC5.10 Week 1+3 - Outside - Node Limit 5% ................. 101
TC5.11 Week 1+3 - Outside - Node Limit 10% ............... 101
List of Algorithms

3.1 Rewarding a specific TA ........................................ 31
3.2 Penalizing a specific TA ........................................ 32
3.3 Rewarding a specific TA - REVISED ......................... 34
3.4 Penalizing a specific TA - REVISED ......................... 35
3.5 Rule Converge Process ........................................ 37
3.6 Forcing a TA to select an action ............................... 38
3.7 Multiple Rule Generation ...................................... 39
3.8 Increasing the Rule Granularity ............................... 41
3.9 Detecting Anomalies ........................................... 43
3.10 Grouping Possible Attacks ................................. 44
Chapter 1

Introduction

Today, many businesses rely on the Internet as an important source of income. For many, it serves as a key channel for advertising as well as internal and external communication services. In addition to this, many businesses depend on services provided on the Internet to carry out their daily work.

As a consequence of its size, the Internet has attracted many malicious users* that may see the vast number of users as an opportunity for dishonest profit. To name an example, such users might be capable of attacking the computer networks to their target companies, which could leave them without Internet connectivity for hours, days or even weeks. Because of the cornerstone position the Internet has adopted in many companies the past two decades, the consequences of such attacks might be devastating.

* In this context, a malicious user is a person who exploit weaknesses in computer software for personal gain, or otherwise partakes in the distribution or creation of malicious software, such as trojan horses, which are non-replicating computer programs planted illegally in another programs that might do damage locally when the software is activated.
1.1 Background and Motivation

Between 1995 and 2003, the Computer Emergency Response Team (CERT) [1] reported an almost exponential growth in reported security incidents, as shown in Figure 1.1.

![Figure 1.1: Reported Security Incidents by Year [1]](image)

Given the widespread use of automated attack tools in recent years, attacks against systems connected to the Internet have become so commonplace that CERT stopped providing these statistics as of 2003. However, even though these numbers are quite outdated, Figure 1.1 clearly indicates that automated attack tools are on a constant, if not exponential, increase. When the number of users connected to the Internet grows larger, so does the amount of potential targets. From the point of view of the regular user, this would actually decrease the probability of being attacked. For the attacker however, the probability of finding a computer vulnerable for attack would increase. Thus, conclusions can be drawn to state that it is becoming increasingly important to protect computer systems against such attacks.

Rule Based IDS

Traditionally, the intrusion detection in computer networks is done using rule based network intrusion detection systems (R-NIDS) [2] such as Snort*, where rules, also known as signatures, are manually generated by security professionals to detect threats in the network traffic. In general, a signature refers to a set of conditions that characterize intrusion activities in terms of network packet headers and payload contents.

This approach relies on a database of attack signatures, and triggers an alarm when one or more of these signatures match what is being observed in the live traffic. Besides lacking the ability to detect novel attacks, a drawback of R-NIDS is that the number of signatures increases along with the number of threats, with the potential of becoming a scalability issue over time.

* [http://www.snort.org](http://www.snort.org)
Using System Dynamics, this problem, hereby referred to as Rule Entropy, can be modeled as an "out-of-control" System Archetype∗, and is shown in Figure 1.2.

If R-NIDS is used to detect new threats in a given environment - new rules are constantly added to compensate for the hostile traffic. This should lower the number of undetected threats, but it can also pose an undesired side effect; as new rules are added over time, the system might enter a state of rule entropy - where the resources required to analyze packets, based on the number of rules, increase. Thus, when the total number of rules reaches a certain level, it takes increasingly more time to manage and delete obsolete rules, leaving the R-NIDS operator with less time to deal with novel attacks.

Anomaly Based IDS

An alternative IDS scheme, known as Anomaly Based NIDS (A-NIDS) is focusing on detecting computer intrusions and misuse by monitoring system activity and classifying it as either normal or anomalous. Since this process does not

---

∗ A system archetype is a variant of the Causal Loop Diagram (CLD). For further reading on such diagrams, the reader is referred to http://en.wikipedia.org/wiki/Causal_loop_diagram.
require a set of pre-defined rules like R-NIDS, A-NIDS possesses the ability to detect novel attacks.

By itself, A-NIDS is not a better solution than R-NIDS [2], as one of its main drawbacks is the number of false positives (FP) generated in current systems, as opposed to real positives (RP). Still, it is a valuable tool for a security analyst as it helps detecting behavior like:

- Hosts that start transmitting abnormal amounts of TCP packets to a foreign (and previously unknown) server. This might indicate that the hosts are infected by malicious software that reports data back to their command center.

- Too many UDP datagrams compared to TCP, which can reveal a misconfigured server or possible denial of service (DOS) attacks towards a local DNS server.

- Port scans from both external and internal hosts.

**Current A-NIDS techniques**

In Anomaly-based network intrusion detection: Techniques, systems and challenges, García-Teodoro et. al. [3] review the most well-known anomaly-based intrusion detection techniques – in addition to presenting systems under development, available platforms and current research projects in the area.

The current techniques can be divided into three main categories: statistical, knowledge-based and machine learning-based. Statistical models usually determine normal network behavior by comparing recent and historical attributes [4], such as bandwidth usage and hosts that communicate with each other – while knowledge-based A-NIDS techniques try to capture the claimed behavior from available system data (like protocol specifications, network traffic instances, etc.) [3]. Finally, machine learning schemes are based on the establishment of an explicit or implicit model, that allows for analysis and categorization of patterns.

Using machine learning in combination with pattern recognition is particularly interesting, as the domain contains areas that still remain unevaluated with regards to A-NIDS.

The next section describes the problem at hand, before Section 1.3 continues by presenting three A-NIDS approaches that exist in the literature.

### 1.2 Problem and Research Questions

The header of each datagram in the Internet Protocol (IP) consists of unique fields that contain information about both its addressing and its contents. By collecting a substantial amount of such packets, one might discover patterns
among them – such as common source addresses, variations in TTL values, uncommon port numbers and so on. By creating a customized computer program for detecting these patterns, it is believed that any underlying semantics in the network packets can be detected, such that normal behavior can be modeled as a data structure.

Frequent Pattern Mining and Association Rules

This thesis describes a data mining scheme for adaptively building Intrusion Detection models that rely on frequent itemsets. Frequent itemsets play an essential role in many data mining tasks that try to extract interesting patterns from databases. Association rules [5], originally defined by Agrawar et. al. [6] for discovering regularities between products in large scale transaction data, is a technique that can be used for this purpose. For example, the rule \{Protocol = ”TCP”, DestinationPort = ”80”, SourceAddress = ”10.0.0.2” \Rightarrow ”normal”\}, that might be found in a set of network packets, indicates that packets with those specific properties would pass as normal behavior.

The problem then becomes how to proceed. The process of modelling normal network behavior, by extracting patterns from a set of IP packets is not completely new, as several methods [7, 8, 9, 10] already implement such approaches. Although A-NIDS have been around for several years, and the techniques are continuously evolving, there still exists several open issues and challenges [3] regarding these systems. In particular, these are related to low detection efficiency and low package throughput because of the required processing power.

In their survey [3], García-Teodoro et. al. mention several systems that use concepts and approaches found in the domain of machine learning, but none of these seem to be related to the Learning Automata paradigm.

Learning Automata (LA)* are adaptive decision making devices that have the ability to operate in both unknown and non-deterministic environments [11]. One of their powerful properties is that they progressively improve their performance through a reinforced learning (RL) process. In addition, they combine fast and accurate convergence with low computational complexity, and have been applied to a broad range of modeling and control problems. [12]

Research Questions

By applying the LA paradigm to mine the frequent itemset patterns, the work presented in this thesis will investigate the scheme’s potential to classify unknown traffic as normal or anomalous.

In essence, the proposed scheme, hereby referred to as the Grimstad Data Classifier (GRIDAC) will attempt to generate rules based on frequent patterns

* Learning Automata are explained in more detail in Section 2.2 on page 23.
in the network packets, without any human supervision. These patterns will be detected by randomly selecting a packet from a dataset, hereby known as the *filter packet*, which contain a certain set of properties. Then, packets with similar properties will be grouped together by LA, such that a rule can be generated which match these properties. This process continues until the dataset is fully covered. The generated rules will be hierarchically organized as a tree data structure, such that their granularity will increase along with the current level in the tree. These steps form the basis for researching the following questions.

**RQ 1** By applying the LA paradigm, is it possible to mine network packets for frequent patterns, such that rules for modeling "normal" behavior can be generated?

**RQ 2** How good is GRIDAC at detecting anomalies, compared to an existing solution? Also, to what extent are false positives\(^*\) and false negatives\(^†\) generated?

**RQ 3** Would the A-NIDS implementation of GRIDAC be able to replace current R-NIDS implementations like Snort?

The next section follows up on *Current A-NIDS techniques*, mentioned on page 4. In particular, the systems NETAD [8], fpMAFIA [10] and MINDS [9] will be reviewed.

### 1.3 Literature Review

During the past decade, there has been much interest in applying pattern recognition and data mining techniques to NIDS, as malicious network traffic often differs from benign traffic in ways that can be distinguished without knowing the nature of the attack [8]. To give an example, Matthew V. Mahoney proposes a system which flags suspicious packets based on unusual byte values in network packets.

#### 1.3.1 Packet Header Inspection

This system attempts to separate normal traffic from hostile traffic, and provide alerts to the system operator. Initially, this is done by identifying five types of anomalies in hostile traffic, and give scores based on how "malicious" the traffic is. These five types of anomalies are [13]:

\(^*\) A false positive occurs when a network packet is inaccurately classified as anomalous, when it is indeed normal.  
\(^†\) A false negative is used to define a malicious packet which is categorized as normal, when it is in fact anomalous.
**User behavior.** Hostile traffic may have a previously unknown source address because it comes from an unauthorized user of a restricted (password protected) service. Also, probing applications such as **nmap** may attempt to access nonexistent hosts and services, generating anomalies in the destination addresses and port numbers.

**Bug exploits.** Attackers usually exploit errors in target software, like heap based buffer overflow vulnerabilities. Such errors are likely to be found in the least-used features of the program, as the error would otherwise been detected during ”normal” use.

**Response anomalies.** Sometimes a target will generate anomalous traffic in response to a successful attack, for example, a victim might send a response to a C&C (Command & Control) server indicating that a trojan is installed on the victim’s computer, and is ready to accept commands from the attacker.

**Bugs in the attack.** When an attack is performed, the client protocols must typically be implemented by the attackers themselves. Due to possible carelessness, or because it is not necessary, the client protocol does not match the protocol standards implemented in benign software. An attacker may use lowercase for convenience, even though normal clients always use uppercase.

**Evasion.** Attackers may deliberately manipulate network protocols to hide an attack from an improperly coded IDS. Such methods include IP fragmentation, overlapping TCP segments that do not match and deliberate use of bad checksums to name some.

Given the variety of anomalies, it makes sense to examine as many attributes as possible. The idea is that if an attribute takes on a novel value, or at least one not seen recently, then the data is suspicious.

The proposed system, Network Traffic Anomaly Detector (NETAD) is based on PHAD (Packet Header Anomaly Detection) [13], also by Mahoney et. al.

### 1.3.2 PHAD and NETAD

PHAD uses time-based models, in which the probability of an event depends on the time it last occurred. For each attribute, a set of allowed values is collected, and novel values are flagged as anomalous. Specifically, a score of $tn/r$ is assigned to a novel valued attribute, where

- $t$ is the time since the attribute was last anomalous (during either training or testing),
- $n$ is the number of training observations, and
$r$ is the size of the set of allowed values.

NETAD shares the same concept as PHAD, using time-based models. There are also some significant differences, like:

1. The traffic data is filtered such that only incoming server requests are examined.
2. Starting with the IP header, only the first 48 bytes are treated as an attribute for the model.
3. The anomaly score $tn/r$ is modified to score rare, but not necessarily novel, events.

To make it easier to detect anomalies, NETAD separately models nine subsets of the filtered traffic corresponding to nine common packet types, such as:

1. All TCP ACKs to port 23 (telnet)
2. All TCP ACKs to port 25 (SMTP)
3. All TCP ACKs to port 21 (FTP)

Essentially, NETAD is a two stage anomaly detection system for identifying suspicious traffic. The first stage filters the input data and generates the model, while the second assigns anomaly scores to unclassified network packets.

For each of the 48 collected attributes, a set of allowed values are generated (i.e. anything observed at least once during the training phase). Then, if one of the attributes contains a value not previously observed, the specified packet is marked as anomalous. This process can be described with the following boolean expression:

$$\{x \lor y \lor z\} \land \{p \lor q \lor r\} \land \cdots \land \{u \lor v \lor w\}$$

Figure 1.3: NETAD Attribute Model as a Boolean Expression

The final result was tested against the 1999 DARPA IDS Evaluation Sets - and it was concluded that this system detects 132 of 185 attacks, with 100 false alarms.

By taking Mahoney’s research into consideration, it is reasonable to adopt the same limitations with respect to the network traffic. As a result, the scheme proposed in this paper will focus on analysing the first 48 bytes of a network packet, starting with the IP header.

As Mahoney’s approach is slightly customized for detecting the attacks in the 1999 DARPA IDS Evaluation Sets, due to his use of 9 different data models - applying unsupervised anomaly detection in NIDS is a new research area that
have already drawn interest in the academic community. In 2005, Leung et.
al. [10] investigates a new density- and grid-based clustering algorithm which
relies on mining frequent itemsets, that is suitable for unsupervised anomaly
detection.

1.3.3 Unsupervised Anomaly Detection in Network Intrusion De-
tection Using Clusters

In [10], Leung et. al. propose a clustering algorithm known as \textit{fpMAFIA}. The
algorithm takes as input a set of unlabeled data and attempts to find intrusions
contained within. After these intrusions are detected, it is possible to train a
misuse detection algorithm or a traditional anomaly detection algorithm using
the data. Although they focus primarily on clustering techniques, mining
frequent itemsets is one of the intermediate steps in their algorithm.

Apparently, \textit{fpMAFIA} is based on the frequent-pattern growth (FP-growth)
algorithm that is quite efficient for mining frequent itemsets [10]. It avoids the
cost of generating a huge set of candidate itemsets, like the well-known Apri-
ori algorithm, by building a compact prefix-tree data structure, the \textit{frequent-
pattern tree} (FP-Tree).

[10] explains that FP-Growth first scans the database, and derives the set of
frequent items and their support (frequency) counts. Then, the set is sorted
in the order of descending support count. To construct the FP-Tree, let \( L \)
denote the resulting set, rescan the database and process the items in each
record in \( L \) order (i.e., sorted according to descending support count). The
processed items should then represent a branch in the tree, with each frequent
item represented by a node. Following, the branch is added to the tree if it
does not exist. If any prefix of the branch already exists in the tree, then
increment the count of each node along the common prefix by one and extend
the branch.
fpMAFIA is an optimized version of the pMAFIA algorithm, with the modification that FP-Tree is used in the intermediate step, and is able to run with a large dataset of 1 million records on a single PC, and terminated in under 11 minutes.

Their algorithm was evaluated using the 1999 DARPA IDS Evaluation Sets, where it was able to achieve a performance rate of 0.867, as shown in the ROC (Receiver Operator Characteristic) chart in Figure 1.4.

![ROC curve of fpMAFIA](image)

The performance rate is calculated as the Area $R$ under the ROC chart. It should also be noted that this particular chart does not show the amount of total attacks that have been detected (i.e. the detection rate), but the rate of detected attacks with regards to the false positive rate. Leung et. al. [10] does not provide the total detection rate.

Their evaluation shows that the accuracy of their approach is close to that of existing techniques reported in the literature [10], and that is has several advantages in terms of computational complexity.

The scheme presented in this thesis also relies on mining frequent itemsets, but it does not implement any of the algorithms that are known from the literature. Instead, it relies on a team of Learning Automata for building a rule tree, similar to FP-growth.

A somewhat different approach, called the Minnesota Intrusion Detection System (MINDS) [9] uses a suite of data mining techniques to automatically detect attacks against computer networks and systems.

### 1.3.4 MINDS - Minnesota Intrusion Detection System

Unlike NETADS, MINDS [9] depends on Netflow version 5 data as input, where the difference to regular network packets is that flow data only capture packet
header information (i.e. it does not capture message content), and build one way sessions (or flows).

Before any data is fed to the anomaly detection module, a data filtering step is performed by an analyst to remove trivial network traffic. Following, the first step in MINDS is extracting features that are used in the data mining analysis, like IP addresses, source and destination ports, protocol type etc. and derived features include calculation of time and connection windows. These features are constructed to capture connections with similar characteristics in the last $T$ seconds.

The following figure gives a general overview of MINDS’s architectural design.

![Diagram](https://via.placeholder.com/150)

Figure 1.5: A general overview of MINDS’s architectural design. [9]

Once the feature extraction step is completed, the known attack detection module is used to detect network connections that correspond to attacks for which signatures are available, and then to remove them from further analysis. The remaining data is fed into the anomaly detection module that assigns anomaly scores to each network connection, and the human analyst may then inspect the most anomalous connections, to determine if they are real or false positives.

Continuing, the association pattern analysis module summarizes network connections that are ranked highly anomalous by the anomaly detection module. Finally, the analyst provides a feedback after analysing the created summaries – and decides whether these summaries are helpful in creating new rules that may be used in the known attack detection module.

Although MINDS was a hybrid of R-NIDS (due to the Known Attack Detection Module) and A-NIDS, the results from their anomaly detection approach are quite satisfactory. In addition, Ertöz et. al. [9] state that it is suitable for detecting many types of threats, such as outsider attack, insider attack, and worm/virus detection after a machine has become infected and starts communicating with its command and control server.

The most interesting aspect from MINDS, in terms of the approach presented in this thesis, is its architectural design. Similar to NETAD, MINDS also make use of anomaly scores, making it reasonable to adopt this feature as well.
With NETAD, Mahoney used the 1999 DARPA IDS Evaluation Sets for evaluation. One of the many original criticisms of this dataset [14], was that it did not evaluate traditional R-NIDS like Snort. Brugger et. al. wanted to do something about this, and they performed an assessment of the DARPA IDS Evaluation Dataset with Snort in 2007.

1.3.5 An Assessment of the DARPA IDS Evaluation Dataset using Snort

In [14], Brugger et. al. performed an evaluation of the 1998 DARPA dataset using the de-facto R-NIDS Snort. Initially, they thought that Snort would perform well on the DARPA dataset, but their empirical results showed the exact opposite.

They discovered that the overall detection performance was low, and that the rate of false positives was unacceptable. At first, they assumed it was due to a failure in the DARPA dataset, or that the attacks were outdated since they used a Snort signature database from 2005. Eventually, they figured that the DARPA dataset only includes a limited number of attacks that are detectable with a fixed signature. Apparently, the majority of the malicious connections present in the 1998 DARPA dataset came from Denial of Service attacks. While Snort has some capability for detecting such attacks, they have not been the primary focus of its design.

For that reason, they do not endorse changing Snort to detect Denial of Service attacks, but rather use Snort in conjunction with another NIDS, designed for such purposes.

1.4 Method

GRIDAC is based on the LA paradigm, found within the Machine Learning domain. In order to get satisfactory answers to the research questions on page 5, it is important to gather observable and measurable evidence through a series of tests, formalized in a Test Programme*. To achieve this, a quantitative approach will be taken, and the collected data will then be used to discuss the final outcome.

In addition to presenting some existing research methods that have been adopted to create and test GRIDAC, the scheme itself is explained briefly in the next sections.

1.4.1 Solution Approach

GRIDAC features two separate stages for classifying binary formatted data. At first, it is necessary to create a hierarchy of rules that will model normal

* Use of the Test Programme is explained in Section 1.4.4 on page 18 and presented in Section 4.1 on page 53.
data. Then, unknown traffic is compared to the model, and it is classified as normal or anomalous. To generate the initial rule, the input data is passed through a feature selection process\textsuperscript{*} – in which one determines what needs to be measured in order to accurately classify objects into distinct classes.

The IP header in network packets contains several features (or bytes) that can be used for this purpose. As an example, packets sent to host A and B from host C can be split into two separate groups based on the bytes that make up the destination IP address, but there are also other fields in the IP header that can be used for the same purpose.

Feature Selection Process

To detect these fields, in an unsupervised manner, a set of split criteria, referred to as a \textit{rule}, must be generated. This rule will then be used to divide the input data into two or more classes. An abstract illustration of this process, given successive trials, is shown in Figure 1.6.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1.png}
\caption{Abstract selection of split criteria. The split criteria used in each rule describe a given class, and reduces the overall entropy in the dataset.}
\end{figure}

A team of LA will generate the rule used in the initial classification process. Each feature will first be subjected to the jurisdiction of a dedicated LA. Next, a random packet will be drawn from the input data, known as the \textit{filter packet}. Reinforced learning (RL) will then be applied to guide the LA towards one of two possible actions; \textit{constant} (C) or \textit{wildcard} (\textasteriskcentered). Here, \textit{wildcard} means that

\textsuperscript{*} Explained in brief in the next section, and in more detail in Chapter 3.
a feature can take on any value, while \textit{constant} requires a feature to have a specific value. 

When all of the LA have converged towards an action, the actions are translated into split criteria and added to a classification rule. An example of a possible rule that could have been generated from the input data shown in Figure 1.6 is given in Table 1.1.

Table 1.1: An example rule generated from the abstract dataset shown in Figure 1.6.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{represented by} & \textbf{classified by} \\
\hline
10001001 & \ast C_0 C_0 C_0 \ast \ast \ast C_0 \ast \\
\hline
\end{tabular}
\end{table}

The triangle object, represented by the bitstring 10001001, has been classified using the rule displayed in Table 1.1, by setting a series of constants and wildcards as criteria.

If one constant in the rule had been set to the opposite value, the object shown in Table 1.2 might have been classified rather than the one showed in Table 1.1.

Table 1.2: Another example rule generated from the abstract dataset shown in Figure 1.6

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{represented by} & \textbf{classified by} \\
\hline
00011100 & \ast C_0 C_0 C_1 \ast \ast \ast C_0 \ast \\
\hline
\end{tabular}
\end{table}

Similarly, if this constant had been set to a wildcard, both objects might have been classified by the same rule, as shown in Table 1.3.

Table 1.3: Example rule that classifies the objects shown in Table 1.1 and Table 1.2

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{represented by} & \textbf{classified by} \\
\hline
10011001 & \ast C_0 C_0 \ast \ast \ast C_0 \ast \\
\end{tabular}
\end{table}

To give a summary of GRIDAC up to this point, it generates a rule that consists of a certain number of features, specified by constant $C$ or wildcard $\ast$. 
Each feature is formulated in terms of operands in a sequence of boolean AND operators, illustrated in Equation 1.1. Specifically, these features are learned by a cooperative game between the LA that aims to divide the dataset in a given ratio.

\[
(f_1 = \alpha_x(u)) \land (f_2 = \alpha_y(v)) \land \cdots \land (f_n = \alpha_z(w))
\]  

(1.1)

However, dividing the dataset is only the first part of the process.

**Increasing Rule Granularity**

The next step attempts to model approximately 100 % of the dataset by selecting multiple filter objects that generate multiple rules. In turn, the granularity of these rules are increased by hierarchically structuring them. This will be done by repeating the same process on the data that is classified by the initially generated rules, as illustrated by Figure 1.7.

![Hierarchical structuring](image)

**Figure 1.7: Hierarchical structuring**

Successive progression will be used at each level in order to classify exactly 100 % of the objects in the dataset, meaning that the data not classified by rule \( n \) should be used for generating rule \( (n + 1) \) and so on.

An attempt will also be made to increase the granularity of each rule at any given level, until a specified limit has been reached that states that the rule should not be able to match less than \( x \% \) of the dataset.
Applying GRIDAC as an A-NIDS

Once the hierarchy of rules has been generated, it is believed that it can be used to model normal behavior in a set of network packets. Thus, when new traffic is introduced, it should to some degree be classified as anomalous – depending on a series of different factors. These factors might include how far an unknown packet is able to traverse the hierarchy, and how many features of an unknown packet that matches a given rule.

While classifying unknown packets, a specified amount of features in all objects present in the dataset will be compared to the generated rule. If all features of a given object matches the requirements of the rule, it will be classified as accepted for that given node in the hierarchy. If exactly one or more of the object’s features deviates from the requirements specified in the rule, it will be classified as rejected – but not necessarily anomalous. For this reason, an anomaly score will be calculated, based on the factors mentioned in the previous paragraph.

To verify that the designed scheme works as intended, a prototype will be developed. This prototype will then be subjected to a formal Test Programme, and the results gathered from these tests will be used to form a conclusion.

1.4.2 Software Development Approach

The process involving the development of the prototype will, as closely as possible, follow industry practice. This is to maximize the probability that minimum standards of quality are being attained in the development of the prototype.

To increase the probability of successful results while dealing with software development, the Guidelines for Secure Software by Futcher et. al. [15] suggests that the development process should structured, planned and controlled from the start - while at the same time using good practices to increase efficiency. [15] also explains that:

- Software should be developed iteratively.
- Requirements should be managed.
- The use of component-based architectures is recommended.
- Software should be modelled using visual abstractions.
- Verification of software quality is important.

The advice from Futcher et. al. lead to the decision of using an iterative software development approach known as Prototyping, first proposed by the US Department of Health and Human Services.[16]

16
This basically implies that the system requirements are defined while the system is being modeled and programmed, as shown in Figure 1.8. The reason why this method was chosen was the possibilities made available by not locking the process from the beginning. By using this method, the thought process can constantly be stimulated, which could lead to new ideas during the development process. This is because the requirements initially defined are likely to change while working on GRIDAC, because of experiences, functionality and design.

According to [16], prototyping is a software development method that is especially useful for resolving unclear objectives. It can also be used for identifying and validating user requirements.

By using prototyping, a set of requirements is expected to be formally defined once the solution has become operational. During this process, the larger components that make up the prototype will be defined and documented in various work packages.

These work packages will contain a description of the system component it is intended for, along with a set of specific requirements. Once the different work packages have been designed, they will be assigned to the team members for execution.

The main idea is that, by separating a large project into smaller, more manageable parts, it will be easier to keep track of how much work is remaining. Once all the work packages have been completed, the prototype should be ready for verification through intensive testing.

1.4.3 Choice of Programming Language

Python has been selected as a suitable programming language, mainly because of how productive a programmer is able to be in a given time frame, compared with other languages like C, C++ or Java.

According to an article on the pros and cons of Python, [17] it is said that "an experienced programmer can probably pick up the basics of Python in a day, be productive in a week or less, and be relatively expert in the language many times faster than she could achieve equivalent fluency in C, C++, Java, or even Perl."
Python is also very strict when it comes to clean code syntax. Unless the code is properly indented, the Python interpreter refuses to compile it. This forces the programmer to write clean code, making it more understandable for fellow programmers.

1.4.4 Quality Assurance

Quality assurance is a collective term for various procedures that are used to both preserve and increase the quality in software projects. The work presented in this thesis tries to adhere to best coding practices, code inspection and use of a formal Test Programme.

Best Coding Practices

To increase the readability of the produced code, it is important to write in a structured and understandable manner. Commenting is also considered an important aspect. In addition, it is important to adhere to the recommended coding conventions of the selected programming languages.

GRIDAC will be programmed, and implemented, using the Python programming language. Therefore, it seems reasonable to adhere to PEP 8* and PEP 257†, which contain guidelines and conventions for programming style and in-line documentation.

Code Quality and Work Package Execution

To ensure a certain standard of quality in the code, thus avoiding software entropy (also referred to as code rot), systematic monitoring and evaluation of the various aspects of the project has been carried out.

As an example, upon completion of a specific work package, it will be queued for inspection. The other team members will then review the code that has been produced, as suggested by Futcher et al. [15].

When the code has been verified, it can be fully tested using the Test Programme.

Test Programme

When the prototype is finished, it will be verified through the use of a formal Test Programme (TP), and help validate that it works according to the specified requirements. The TP includes a set of test cases, carefully designed to test different aspects of both GRIDAC and the prototype it will be implemented in. By executing these tests, the need for design changes can be minimized.

* Style Guide for Python Code: http://www.python.org/dev/peps/pep-0008/
† Docstring Conventions: http://www.python.org/dev/peps/pep-0257/
Once the formal Test Programme has been verified, the achieved results and findings will be discussed and the work will be concluded.

### 1.5 Key Assumptions and Limitations

The novel scheme for modelling normal traffic patterns, and for detecting anomalies in binary formatted data, will be implemented as an A-NIDS prototype. This prototype will consist of several components, and the field of research is quite vast. Thus, to provide a basic framework for research, it has been necessary to apply some key assumptions and limitations.

#### Assumptions

- **Prior Research and Data is Correct**
  It is assumed that the previous research mentioned in Section 1.3 is correct, and that the results are valid and reproducible. Also, it is assumed that the number of computer related attacks, based on the amount of reported incidents as shown in Figure 1.1 on page 2, are still on a constant, if not exponential, increase.

- **DARPA IDS Evaluation Sets**
  When GRIDAC is implemented as an A-NIDS, it will be evaluated with the 1999 DARPA IDS Evaluation Sets [18]. Even though the specific attacks in this dataset are outdated, it is assumed that current attacks stand out in a similar way, such that they can be detected as anomalies by GRIDAC. Although, the payload in current attacks (like SpyEye *) are in most cases encrypted, the IP header still remains in cleartext (unless VPN technologies like IPSec are used).

- **Attack Traffic is Statistically Different**
  It is assumed that the attack traffic is statistically different from normal traffic. Hence, traffic that deviates from the normal traffic patterns might indicate a possible attack.

In addition to these assumptions, the following limitations further narrows down the scope of the work being done.

#### Limitations

- **Inspected Bytes of each Packet**
  Similar to NETAD, only the 48 first bytes of the network packets are analyzed, starting with the IP header.

• **Hardware Support**
  Hardware support will be limited to the 32-bit and 64-bit compatible x86-platform, more specifically i386 and upwards in addition to x86_64.

• **Software Platform**
  The platform used for testing and verification will be based on GNU/Linux. It is not within the scope of this thesis to make it work on other software platforms.

• **Performance**
  Performance, in terms of time consumption during the modelling and classification process, is assumed to be of less importance than the actual outcome, and is thus out of scope.

• **Classification of DARPA IDS Evaluation Sets**
  Because of time limitations, only the attacks categorized as ”outside” will be analyzed. These attacks are listed in the DARPA IDS Evaluation Set Detection Truth lists. * †

The next section presents what contributions the work done in this thesis will add to current knowledge within the chosen field of research.

### 1.6 Contribution to Knowledge

The work presented in this thesis investigates if LA can be applied to mine frequent items from a dataset, such that it can be modeled as rules organized in a tree structure. When unknown, but similar, traffic is introduced to the tree structure, most of the objects should be classified as normal, while others are reported as anomalous - mainly because they do not relate to the model. As such, the work also examines if the aforementioned approach can be applied to A-NIDS scenarios by detecting anomalies in a set of network packets.

A prototype will be created, and it will be evaluated with the 1999 DARPA IDS Evaluation Sets [18], and the results will be compared towards those of NETAD, presented in Chapter 1.3.

### 1.7 Thesis Outline

The rest of this thesis is organized as follows: Chapter 2 contains theory on Machine Learning, and explains how it can be applied to make decisions in non-deterministic environments. Information on Learning Automata is also given, explaining what these devices can be used for, before taking a look at one of the many LA implementations, known as a Tsetlin Automaton. Also,

---

† [http://www.ll.mit.edu/mission/communications/ist/files/master_identifications.list](http://www.ll.mit.edu/mission/communications/ist/files/master_identifications.list)
a short summary of patterns in TCP/IP is given in order to make the reader better understand the key concepts behind GRIDAC. This should provide the reader with enough information to be able to take in the finer details of the proposed solution.

In Chapter 3, the solution approach is explained. A set of requirements and design guidelines for the system has been given, and models of the key components are added. As a whole, the solution chapter provides solid documentation of all the important aspects of the designed system.

System verification and testing is documented in Chapter 4. Here, results from the various tests, carried out in a formal Test Programme, is presented, and the research questions are also investigated.

Chapter 5 is used to discuss the results that were obtained in Chapter 4. Problems with the proposed solution is brought to attention, and an effort has been made in order to identify their causes. It is also determined if the results from Chapter 4 are valid, and if the solution is correct. In some cases, steps for dealing with the identified problems are also given.

Finally, Chapter 6 provides a brief summary of the solution that has been developed. The main findings, and corresponding implications are shown. Lastly, options for future work are given.

The Appendices include an example report taken from the formal Test Programme and the Work Package Overview, respectively.
Chapter 2

Machine Learning and Applications

A dictionary defines learning as a modification of behavioral tendency by experience. In his book Introduction to machine learning [19], Nils Nilsson draws parallels between machine learning and animal training, where the behavior of the system (or the animal) is modified by rewarding good decisions and punishing bad decisions.

2.1 Reinforced Learning

The concept of reinforced learning can be illustrated by the well known T-maze learning problem, as shown in Figure 2.1, where a mouse (or an automaton) interacts with a maze, trying to find cheese.

![Figure 2.1: Hungry Mouse in a T-Shaped Maze](image)

Should the mouse decide to go right, it is rewarded - with the overall goal of selecting the same direction in future decisions. Should the mouse decide the opposite (going left), it is punished. With successive trials, the mouse will hopefully learn to make the correct decision.
2.2 Learning Automata

In *Learning Automata: an introduction*, Narendra and Thathachar [11] define Learning Automata (LA) as adaptive decision making devices that have the ability to operate in both unknown and non-deterministic environments. This implies that they are able to perform tasks without any information about the effect of their actions at start of an operation - and that a given action not necessarily produce the same response each time it is performed.

According to [11], one of the powerful properties of LA is that they progressively improve their performance through a reinforced learning process - similar to the T-Maze problem (illustrated in Figure 2.1), where the mouse interacted with a specific environment.

In general, an environment is a large class of unknown media in which an automaton or a group of automata can operate, or perform actions. Once an action is performed, the environment responds with either a penalty or a reward, as shown in Figure 2.2.

![Figure 2.2: Relation between actions and rewards, with regards to the environment.](image)

More specifically, the automaton can perform an action, $a_i$, from a set of unique actions, $a_1, a_2 \ldots a_i$. When performing the action $a_i$, there is a certain probability that the environment responds with a penalty.

$$P(\text{Penalty}|\text{Action} = a_i) = c_i, 1 \leq i \leq r$$  \hspace{1cm} (2.1)

The responses from the environment are in turn used as input to the automaton, which maps it to its internal ‘memory’ - such as a series of different states. When the current state of the automaton is updated, new input is received from the environment, and the automaton is learning by reinforced measures. This process can be implemented in different ways, and one of these is the Two-action Tsetlin Automaton.
2.3 Tsetlin Automaton

A Two-action Tsetlin Automaton (TA) [20] operates with two different actions, such as yes or no, or true or false. An example of a TA is shown in Figure 2.3.

![Diagram of Two-action Tsetlin Automaton with 3 states per action.]

Figure 2.3: Two-action Tsetlin Automaton with 3 states per action.

It comprises of \( n \) states per action, meaning that for each answer it is able to provide, it maintains an internal 'memory' of \( n \) different states. Once an answer is given, the automaton is either rewarded or penalized. If it keeps giving the same answer over and over, the current state of the automaton is incremented towards either end state.

When the current state reaches state 1 or state \( n \), the automaton has converged.
2.4 Patterns in Network Traffic

Network protocols are a formal description of digital message formats and the rules for exchanging those messages between computing systems. Because of these strict message formats, it is possible to investigate a set of network packets with the purpose of identifying similarities or differences between them.

As seen in the below figure, the IPv4 packet header consists of 14 fields, of which 13 are required [21]. Obviously, the optional field is "options", which is rarely in use. The fields in the header are packed with the most significant byte first (big endian), meaning that the hexadecimal notation of \(0x0001\) equals 1 in decimal notation.

![IPv4 Packet Header Structure](image)

Some of these fields remain constant in certain contexts. For example, if tcpdump\(^*\) or Wireshark\(^†\) is used to analyze the network traffic originating from host A towards host B, the IP Source and Destination Address field would most certainly remain constant (given that one direction is analyzed, and not both). Also, if IPv4 is the only network layer protocol in use, the IPv4 version field would similarly remain constant.

There are also fields in the IPv4 header that seldom are in use. For this reason, they are either set to their default value or 0 – and can thus be regarded as constant. A good example of such fields are the IPv4 Header Length and the Options field.

On the other hand, an example of a field that almost never is constant is the Header Checksum field.

\(^*\) http://www.tcpdump.org/  \(^†\) http://www.wireshark.org/
Chapter 3

Solution

The following sections will give an in-depth presentation of GRIDAC. Based on a list of requirements and design guidelines associated with the scheme, its basic aspects will be explained in full detail.

GRIDAC consists of different components, and the component development workload have been divided into different work packages.

This chapter is written in a partial bottom up approach. Once the requirements have been defined, an overview of the various work packages is given before the inner workings of GRIDAC are presented. Then, a study of how GRIDAC can be implemented as an A-NIDS is given. Finally, the graphical user interface of the prototype is presented.

3.1 Requirements

To test GRIDAC in its entirety, a prototype will be developed in the Python programming language. Primarily, it should be able to handle datasets that consists of either network packets or artificial datasets with properties similar to network packets.

The following lists define the requirements of the prototype, in addition to certain design guidelines. Each of these requirements and guidelines are represented with an ID, a short title, and a more explanatory description.

The requirements describe certain technical features of what GRIDAC is supposed to accomplish. These are:

REQ 1 Customization of Experiments

It must be possible to toggle and edit various attributes in order to tailor the system for a specific experiment, such as:

- Specifying how much data each tree node should try to match, e.g. 50%.
• Setting the number of iterations before deciding on a given rule (i.e. how many runs that should be performed before selecting the best matching rule).
• Restricting the amount of bits each rule element should represent (e.g. 8 bits of data equals 1 element).
• Specifying the maximum tree depth and node limit.
• Increasing the amount states per action for each TA.
• Choosing the amount of randomly selected packets for the training period.

REQ 2 Use of Real Network Traffic

The prototype should support reading raw network packets from a given interface, and also from the capture file format libpcap*.

REQ 3 Use of Artificial Datasets

The prototype should support reading artificial datasets which are binary structured. This makes it easier to interpret how GRIDAC behaves in different scenarios.

REQ 4 Hierarchical Organization

It should be possible to organize the generated rules hierarchically, such that their granularity will increase along with the current level in the hierarchy.

REQ 5 Graphical User Interface

It should be possible to interact with the prototype using a graphical user interface (GUI). The GUI should be able to list possible attacks, a graphical representation of the rule hierarchy, as well as other information that might be of use to the analyst that uses it.

In addition to these requirements, certain design guidelines will also be followed.

Design Guidelines

The design guidelines are rules that should be followed in order to ensure development of good quality code. The following guidelines identifies core principles and best practices to assist in creating the prototype in the best possible manner.

DG 1 Good Coding Practices

All written code must follow good coding practices to ensure code

* For more information about the libpcap format, the reader is referred to http://wiki.wireshark.org/development/libpcapfileformat
cleanness and security. As mentioned in Section 1.4, this would be the Python Enhancement Proposals (PEP) 8 and 257.

A few examples from PEP 8 is:

- 4 spaces per indentation level.
- Maximum line length in the code should be set to 79 characters.
- How existing (and also self-written) libraries should be imported into the code.

**DG 2 Object Oriented Programming**

The prototype must be written in an Object Oriented Programming (OOP) language. By splitting the prototype into different classes and methods, it will make it easier to extend with additional features later on.

With the requirements and design guidelines presented, the work packages can be defined.
3.2 Work Package Overview

The following work packages have been defined based on the aforementioned requirements and research questions. By dividing the workload into different work packages, it will help distribute the workload and also help keep the overall work on track.

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WP 1</td>
<td>Classifier - Basic</td>
<td>The classifier is responsible for detecting split criteria in the input data. This is done by creating a rule that represents a given ratio of the data in question.</td>
</tr>
<tr>
<td>WP 2</td>
<td>Classifier - Hierarchical</td>
<td>Once an initial rule has been generated, the hierarchical part of the classifier takes over, and attempts to generate new rules with similar properties as the parent rule. The only difference is that the new rules are more fine-grained.</td>
</tr>
<tr>
<td>WP 3</td>
<td>Anomaly Detector</td>
<td>When the classification process is complete, and the hierarchy of rules is generated, the next step is to compare unknown data towards the set of rules. This process also introduces the use of anomaly scores that will help distinguish false positives (FP) from real positives (RP).</td>
</tr>
<tr>
<td>WP 4</td>
<td>Graphical User Interface</td>
<td>A GUI will make it easier to use the prototype, and keep track of the results when testing GRIDAC and the anomaly detector.</td>
</tr>
<tr>
<td>WP 5</td>
<td>Graphing</td>
<td>With graphs, it will be possible to give a graphical representation of the rules generated by WP1 and WP2.</td>
</tr>
</tbody>
</table>

During the next sections, the most important aspects from these work packages are presented, such as how GRIDAC functions in detail, and how the anomaly detector is implemented.
3.3 Action Selection

One of the important aspects of unsupervised learning with LA, is determining how each automaton should be rewarded or penalized based on the actions it performs.

As presented in Section 1.4, GRIDAC will implement a feature selection process, with the purpose of creating distinct classes to accurately classify objects. This will be done by randomly selecting an object, also known as the filter object (or packet, if the input data are network packets) from a set of objects, that contain both common and unique features. In turn, the common features will be found by comparing a large amount of randomly selected objects with the filter object.

Using an LA scheme called Tsetlin Automata (TA), GRIDAC will detect these features by assigning a TA to each respective object’s attribute. More specifically, while the features of the filter object are being compared to those of the other randomly selected objects, the TA will decide between two actions; constant \((C)\) or wildcard \((\star)\), and attempt to converge towards either action – depending on whether the compared features of the objects match those of the filter object or not.

The action selecting process will be accomplished by applying reinforced learning (RL). This implies the use of both rewards and penalties in order to make each TA converge. The probability \(r\), where \(r\) is a rational number between 0 and 1, of assigning a penalty or reward will be handled by a governing process that aims to classify a given ratio, \(x\), of the total amount of objects.
3.4 Values of $r$ With Respect to $x$

Considering a simplified set of objects where the possible values of each object’s feature can be either 0 or 1, and that the set is created in a way where 70 % of the objects differ from the remaining 30 %, it is believed that this traffic amount can be matched by creating a rule in which 70 % of the available features are set to match a constant. In Figure 3.1, a TA is shown with equations for selecting action probabilities in such scenarios.

As displayed in the above figure, the probability of giving the automaton a reward, thus incrementing its current state towards $C$, is set to $x \cdot 1.0$, where $x$ in the previous case would be 0.7. To make the remaining 30 % converge towards $\star$, the action probability is set to $(1 - x) \cdot r$, where $r$, will help decrease the probability of selecting $\star$, with the overall goal of making the rule, illustrated in Figure 1.1 on page 15, more accurate.

More specifically, the process of increasing and decreasing the current state of each TA is done using the following algorithms. Algorithm 3.1 shows how a specific TA is rewarded based on the value of $r$.

**Algorithm 3.1 Rewarding a specific TA**

```plaintext
y = the current state
n = the number of states per action
prob = the reward probability
random = a rational number between 0 and 1
if random ≤ prob then
    if y < 0 and y ≥ −n then
        y = y − 1 \{Decrease the current state\}
    else if y ≥ 0 and y < n then
        y = y + 1 \{Increase the current state\}
end if
end if
```

Figure 3.1: Tsetlin Automaton with $n$ states per action and action probabilities.
Similarly, Algorithm 3.2 shows how a specific TA is penalized based on the value of $r$.

**Algorithm 3.2** Penalizing a specific TA

- $y =$ the current state
- $n =$ the number of states per action
- $prob =$ the penalize probability
- $random =$ a rational number between 0 and 1

```
if random $\leq$ prob then
  if $y < 0$ and $y \geq -n$ then
    $y = y + 1$ \{Increase the current state\}
  else if $y \geq 0$ and $y < n$ then
    $y = y - 1$ \{Decrease the current state\}
  end if
end if
```

**Setting $r$ Dynamically**

Once these algorithms were implemented, the accuracy of the LA were quite good, but the time they used to converge towards a rule, was not satisfactory. For this reason, it was believed that the time used for the TA to converge might decrease by considering the aforementioned action probabilities as *forces* that dragged towards $\star$ or $C$, and that the value of $r$ could be dynamically set by letting the action probability for $\star$ be equal to that of $C$, as seen in Equations 3.1, 3.2 and 3.3.

\[
x \cdot 1.0 = (1 - x) \cdot r \tag{3.1}
\]

\[
r = \frac{x \cdot 1.0}{1 - x} \tag{3.2}
\]

\[
r = \frac{x}{1 - x} \tag{3.3}
\]

With $r$ being calculated based on the value of $x$, $r$ becomes larger than 1 if $x$ is set to 0.5 or more, as shown in Table 3.2.

<table>
<thead>
<tr>
<th>$x$</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>0.11</td>
<td>0.25</td>
<td>0.43</td>
<td>0.67</td>
<td>1</td>
<td>1.5</td>
<td>2.33</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 3.2: Values of $r$ when $0 < x < 1$
Markov Chains

Since $r$ is the probability used for giving a reward or penalty, setting the value larger than 1.0 will not have an immediate effect. To add support for values of $r$ larger than 1, a possibility might be to move at least $\lfloor r \rfloor$ states towards either direction for such values of $r$, with an additional probability of $r \mod 1$ for incrementing the current state even further. In probability theory and statistics, this is known as a Markov chain [22]. An illustration of this design is shown in Figure 3.2, where $r$ is set to 2.33.

![Markov Chain Diagram](image)

Figure 3.2: Markov Chain where $r$ is set to 2.33.

Implementing this design resulted in the TA converging faster for large values of $r$, making them more deterministic, while still maintaining the same accuracy. The case was not the same for lower values of $r$, where the LA were more stochastic in behavior, hence increasing the time to converge. To make the TA deterministic for lower values of $x$, a possibility might be to inverse Equation 3.3 when $r < 1 \iff x < 0.5$, resulting in Equation 3.4.

$$r = \begin{cases} x & \text{if } 0.5 \leq x < 1 \\ \frac{1-x}{x} & \text{if } 0 < x < 0.5 \end{cases} \quad (3.4)$$

The following table shows the new values of $r$ with respect to $x$ when Equation 3.4 is used.

<table>
<thead>
<tr>
<th>$x$</th>
<th>0.1</th>
<th>0.2</th>
<th>0.3</th>
<th>0.4</th>
<th>0.5</th>
<th>0.6</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>9</td>
<td>4</td>
<td>2.33</td>
<td>1.5</td>
<td>1</td>
<td>1.5</td>
<td>2.33</td>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

When these values of $r$ were used, the TA also converged faster for lower values of $x$. A side effect was that the accuracy also suffered a small drop, but this could be compensated for by increasing the number of states per action for each independent TA.
Meanwhile, algorithms 3.1 and 3.2 had to be revised to support the markov chains. The revised version for rewarding a specific TA is shown in Algorithm 3.3.

**Algorithm 3.3** Rewarding a specific TA - REVISED

- \( y \) = the current state
- \( n \) = the number of states per action
- \( prob \) = the reward probability
- \( random \) = a rational number between 0 and 1

while \( prob > 1.0 \) do
    if \( y < 0 \) and \( y \geq -n \) then
        \( y = y - 1 \) \{Decrease the current state\}
    else if \( y \geq 0 \) and \( y < n \) then
        \( y = y + 1 \) \{Increase the current state\}
    end if
end while

if \( random \leq prob \) then
    if \( y < 0 \) and \( y \geq -n \) then
        \( y = y - 1 \) \{Decrease the current state\}
    else if \( y \geq 0 \) and \( y < n \) then
        \( y = y + 1 \) \{Increase the current state\}
    end if
end if
Similarly, the revised version for penalizing a specific TA is shown in Algorithm 3.4.

**Algorithm 3.4** Penalizing a specific TA - REVISED

\[
y = \text{the current state} \\
n = \text{the number of states per action} \\
prob = \text{the penalize probability} \\
random = \text{a rational number between 0 and 1}
\]

\[
\textbf{while } prob > 1.0 \textbf{ do} \\
\hspace{1em} \text{if } y < 0 \text{ and } y \geq -n \text{ then} \\
\hspace{2em} y = y + 1 \{\text{Increase the current state}\} \\
\hspace{1em} \text{else if } y \geq 0 \text{ and } y < n \text{ then} \\
\hspace{2em} y = y - 1 \{\text{Decrease the current state}\} \\
\hspace{1em} \text{end if}
\]

\[
\textbf{end while}
\]

\[
\textbf{if } random \leq prob \textbf{ then} \\
\hspace{1em} \text{if } y < 0 \text{ and } y \geq -n \text{ then} \\
\hspace{2em} y = y + 1 \{\text{Increase the current state}\} \\
\hspace{1em} \text{else if } y \geq 0 \text{ and } y < n \text{ then} \\
\hspace{2em} y = y - 1 \{\text{Decrease the current state}\} \\
\hspace{1em} \text{end if}
\]

\[
\textbf{end if}
\]

Now that the possible values of \( r \) have been determined, the next step in the approach can be explained; rule generation.

**Rule Generation**

Once the automaton chooses action \( C \), \( 0.5 \leq x < 1 \Leftrightarrow r < 1 \), and an object passes - it is rewarded with \( r \), as shown in Table 3.4. If the object does not pass, it is given a penalty of \( 1.0 \). If \( * \) is chosen and a constant could have been used instead, the automaton is penalized with \( 1.0 \). If \( * \) was correct, it is rewarded with \( r \).

**Table 3.4: Action probabilities when \( r < 1 \)**

<table>
<thead>
<tr>
<th>Action</th>
<th>Pass</th>
<th>Not Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha_1 ) - *</td>
<td>Correct: ( P(\text{reward}) = r ) \hspace{1em} Incorrect: ( P(\text{penalty}) = 1.0 )</td>
<td>( P(\text{penalty}) = 1.0 )</td>
</tr>
<tr>
<td>( \alpha_2 ) - C</td>
<td>( P(\text{reward}) = r )</td>
<td>( P(\text{penalty}) = 1.0 )</td>
</tr>
</tbody>
</table>

If \( 0 < x < 0.5 \Leftrightarrow r \geq 1.0 \), and the automaton chooses action \( C \), and the object passes - it is rewarded with \( 1.0 \), as shown in Table 3.5. If the object does not pass, it is penalized with \( r \). If \( * \) is chosen, and the action is correct, the automaton is rewarded with \( r \). If \( C \) could have been selected instead, the automaton is penalized with \( r \).
Table 3.5: Action probabilities when $r \geq 1$

<table>
<thead>
<tr>
<th>Action</th>
<th>Pass</th>
<th>Not Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha_1$ &amp; Correct: $P(reward) = 1.0$</td>
<td>$-$</td>
<td></td>
</tr>
<tr>
<td>$\alpha_2$ &amp; Incorrect: $P(penalty) = r$</td>
<td>$P(penalty) = r$</td>
<td></td>
</tr>
</tbody>
</table>

When the training is complete, and all the TA have successfully converged, the overall rule will consist of several independent features, where each distinct feature will tell which action it has converged against and the filter objects’ bitstring for that particular feature.

As an example, if each of the first 48 bytes of a given network packet was treated as a feature, the rule can be expressed in the following boolean statement, where the argument for each action ($\alpha$) represents the bitstring of the filter packet’s feature:

$$(f_1 = \alpha_x(u)) \land (f_2 = \alpha_y(v)) \land \cdots \land (f_{48} = \alpha_z(w))$$

In order for a packet to match (and pass) the rule, it must also match all of the features specified in the rule.
To get a more detailed view of how the classification process in GRIDAC works, the pseudocode in Algorithm 3.5 is provided – which illustrates how the reward and penalty probabilities are assigned to a specific TA, depending on a given action $\alpha$, and how the rule is generated.

**Algorithm 3.5 Rule Converge Process**

```plaintext
object_{filter} = read filter object
r = calculate $r$ based on the given value of $x$

while rule $\neq$ converged do
  object_{random} = read random object from dataset
  compare object_{random} with object_{filter} {Allow the TA to make a decision}
  rule = array of unconverged TAs
  for each TA in rule do
    $\alpha = \text{let TA make decision} \{\text{Depends on the current state of the TA.}\}$
    if $r < 1$ then
      if $\alpha_2$ and object_{random} passes then
        reward TA with $r$
      else if $\alpha_2$ then
        penalize TA with 1.0
      else if $\alpha_1$ and object_{random} passes then
        penalize TA with $r$
      else
        reward TA with 1.0
    end if
  else
    if $\alpha_2$ and object_{random} passes then
      reward TA with 1.0
    else if $\alpha_2$ then
      penalize TA with $r$
    else if $\alpha_1$ and object_{random} passes then
      penalize TA with 1.0
    else
      reward TA with $r$
  end if
  end for
  if all TA have converged then
    rule = converged
  end if
end while
write rule based on TA action values
```

Now that the rule generation process have been presented, the next step in the process is dealing with datasets that contain larger amounts of randomness.
**Action Forcing**

In some cases, the TA might encounter difficulties selecting a specific action due to large amounts of randomness in the dataset. For this reason, it is often necessary to force the TA into selecting a specific action, since it is unable to converge by itself. This step, which in practice is merged into Algorithm 3.5, is explained in Algorithm 3.6.

**Algorithm 3.6** Forcing a TA to select an action

$$TA = \text{a given TA}$$

$$TA_{\text{iteration\_counter}} = \text{iteration counter for } TA$$

$$force_{\text{converge\_limit}} = \text{force converge limit given by user}$$

**if** $$TA_{\text{iteration\_counter}} > force_{\text{converge\_limit}} \text{ then}$$

$$\star_{\text{counter}} = \text{amount of } \star\text{s received.}$$

$$C_{\text{counter}} = \text{amount of } C\text{s received.}$$

**if** $$\star_{\text{counter}} > C_{\text{counter}} \text{ then}$$

force TA to $$\star$$

**else**

force TA to $$C$$

**end if**

**reset all** $$TA$$ **which have not yet converged**

**end if**

When a TA has been forced to either action, the remaining TA which have not yet converged are reset to their default values. The reason for this is that the forced decision of a single TA might have an impact on the game played by the remaining TA, thus potentially creating an obscure rule which does not match the dataset - unless the remaining TA are reset.

This concludes the feature selection process, and the first part of GRIDAC. Using the aforementioned algorithms, the scheme is able to generate a rule that selects a split criterion in a dataset, and splits it in two separate parts, based on the given split ratio $$x$$.
3.5 Multiple Rules and Hierarchical Organization

The next step attempts to classify approximately 100 % of the dataset by selecting multiple filter objects that generate a set of rules, \( \mathcal{S} = \{s_1, s_2, \ldots, s_n\} \), as shown in Figure 3.3. The data not being classified by rule \( s_n \) will be used to converge the TA into generating rule \( s_{n+1} \).

![Figure 3.3: Selection of filter object. The filter object for rule \( s_{n+1} \) is based on the rejected objects from rule \( s_n \).](image)

Pseudocode for describing this process in detail is given in Algorithm 3.7.

**Algorithm 3.7 Multiple Rule Generation**

\[
\begin{align*}
x &= \text{classification target} \\
data\_set &= \text{dataset given by user} \\
\text{rules} &= \text{array} \\
\text{while} \quad data\_set \quad \text{is not classified} \quad \text{do} \\
\quad &\text{generate} \quad \text{rule} \quad \text{by} \quad \text{attempting} \quad \text{to} \quad \text{describe} \quad x \% \quad \text{of} \quad data\_set \\
\quad &\text{with} \quad \text{rule}, \quad \text{calculate} \quad \text{fraction} \quad \text{of} \quad \text{passed} \quad \text{objects} \\
\quad &\text{append} \quad \text{rule} \quad \text{to} \quad \text{rules} \\
\quad &\text{subtract} \quad \text{fraction} \quad \text{of} \quad \text{covered} \quad \text{data} \quad \text{from} \quad data\_set \\
\text{end} \quad \text{while}
\end{align*}
\]

Once the initial rules \( \mathcal{S} \) have been generated, an attempt will be made to increase the granularity of each rule in \( \mathcal{S} \) – by repeating the feature selection process using data that classifies rule \( s_n \), such that a set of rules \( \mathcal{T} = \{t_1, t_2, \ldots, t_m\} \) can be defined as a subset of \( \mathcal{S} \), \( t_m \subseteq \mathcal{S} \). This is because the data classified by any rule in \( \mathcal{T} \) is also classified by the parent rule in \( \mathcal{S} \).
The process of generating the hierarchically organized rules is illustrated in Figure 3.4.

For each of the initially generated rules in $\mathcal{S}$ (corresponding to level 2 in Figure 3.4), new rules, $\mathcal{T}$, will be generated based on the data classified by the parent rule $s_n$, as shown in Algorithm 3.8 on the next page.
Algorithm 3.8 Increasing the Rule Granularity

\[ x = \text{classification target} \]
\[ r = \text{reward for selecting } \star \]

\textbf{for each} rule in rule set do
  \text{data set} = \text{dataset classified by rule}
  \text{newrules} = \text{array}
  \textbf{while} data set is not classified by newrules do
    \textbf{while} newrule is duplicate of parent rule \textbf{or}
    \text{newrule} only contains \star \textbf{or}
    newrule is more general than rule do
      \[ r = r \cdot 0.999 \] \{Decrease the reward for selecting \star.\}
      generate newrule by attempting to describe \( x \% \) of data set
      with newrule, calculate fraction of passed objects
    \textbf{end while}
    append newrule to newrules
    subtract fraction of covered data from dataset
  \textbf{end while}
\end{for}

While the new rules are generated, there is a possibility of selecting an infrequent filter object. If this happens, all the TA might converge towards \star because the remaining traffic is significantly different from the filter object. As such, certain criteria are set while the new rules are generated. If the new rule is a duplicate of the parent rule, only contains \star, or is more general than the parent rule – that particular rule, \( t_m \), will be skipped, and another filter object will be selected.

To further decrease the possibility of selecting an infrequent filter object, the reward for selecting \star with any given TA, is decreased in each attempt.

Once the dataset classified by rule \( s_n \) is covered by \( T \), the process continues with rule \( s_{n+1} \).

When the hierarchy of rules is completed, it should be possible to send unknown objects through the hierarchy, and depending on how similar these objects are to the rules, they will be given an anomaly score. This will be explained in the next section.
3.6 GRIDAC as an A-NIDS

In order to answer RQ 2, a prototype has to be developed that is able to match unknown objects against the hierarchy of rules generated by GRIDAC, and report whether the packet is anomalous or not. The current working name of this prototype is Inspectobot, and its architectural design is shown in Figure 3.5.

![Diagram of Inspectobot's architectural design.](image)

Figure 3.5: Inspectobot’s architectural design.

Similar to the architectural design of MINDS, as shown in Figure 1.5 on page 11, Inspectobot will be able to read packet streams from the network interface. In addition, it should be possible to use pre-generated datasets like the DARPA IDS Evaluation Sets, or more customized (artificially generated) datasets to test each aspect of Inspectobot. Unlike MINDS, Inspectobot will need to enter one of two different modes. If the mode is set to Training, GRIDAC is used for generating rules that are stored in a hierarchy. If the mode is set to Testing, the objects (or network packets) in the dataset is matched against the hierarchy of rules. If the packet is found to be anomalous, it is given an anomaly score, and the metadata of each packet is added (along with the anomaly score) to a relational database. Then, the Security Analyst has the ability to both sort and group the anomalies, based the anomaly score or the packet’s source or destination addresses. This way, it is possible to accumulate the anomaly score of several packets, based on their metadata.
When Inspectobot enters the Testing mode, it will check each packet in the dataset sequentially. The packet is then sent to the anomaly detector where it is tagged as anomalous or normal. This process is explained in Algorithm 3.9.

**Algorithm 3.9 Detecting Anomalies**

```plaintext
for each packet in data_set do
    processed_rules = array
    current_rules = initial_rules {The initial rules generated in the hierarchy}
    while current_rules is not empty do
        pop current_rule from current_rules
        most_general_rule = current_rule
        if current_rule is not in processed_rules then
            if packet matches current_rule then
                current_rules = the children of current_rule {Continue down the hierarchy}
            else if current_rules is empty then
                wildcards = amount of * in most_general_rule
                constants = amount of C in most_general_rule
                anomaly_score = wildcards - constants
                break
            else
                if current_rule contains more * than most_general_rule then
                    most_general_rule = current_rule
                end if
            end if
        end if
        push current_rule to processed_rules
    end while
    if anomaly_score > 0 then
        store anomaly_score in database
        store packet metadata in database {IP addresses, ports etc.}
        store rule metadata in database {features that don’t match the packet}
        store parent_rule id in database
    end if
end for
```

The packet is then matched against the initially created rules (or the root nodes in the hierarchy) in the same order as when the rules were generated. If the packet matches a given rule (or node), it is then matched against the rule’s children (or child nodes). If the packet does not match a specific rule, it is matched against the other rules at the current level in the hierarchy. If the packet does not match a specific rule, it is matched against the other rules at the current level in the hierarchy. If the packet does not match any of the nodes in the current level in the hierarchy, it is flagged as anomalous, and an anomaly score is calculated based on the most general rule at that specific level (meaning the rule with the most *). This way, an anomalous packet receives the maximum amount of points at any given level.
Calculating the anomaly score is not a complex task, as it is merely the difference between the amount of $\star$ and $C$ for the rule in question. However, if the anomaly score is 0, or negative, the packet is classified as normal. The anomaly score may be negative if a packet is able to work its way down the hierarchy at the point where the rules contain more $C$ than $\star$. This might also happen at the higher levels in the hierarchy, as there might exist a rule in one of the top levels that contains more $C$ than $\star$.

In addition to generating an anomaly score, Inspectobot also attempts to group anomalies into possible attacks, by looking at the packet’s address fields and timestamp value, as shown in Algorithm 3.10.

Algorithm 3.10 Grouping Possible Attacks

```plaintext
for each anomaly in detected_anomalies do
    matching_attacks = array
    source_address = anomaly source address
    destination_address = anomaly destination address
    timestamp = anomaly time stamp
    new_attack = false
    for each attack in possible_attacks do
        a_source = array of all source addresses in attack
        a_destination = array of all destination addresses in attack
        a_common = a_source \ a_destination
        if (source_address in a_source or destination_address in a_destination) and
           source_address \ a_common and destination_address in a_common
            push attack to matching_attacks
        end if
    end for
    for each attack in matching_attacks do
        last_timestamp = last timestamp in attack.
        if timestamp - last_timestamp > 3600 then
            new_attack = true
        else
            push anomaly to attack {Update attack with anomalous packet.}
        push attack to possible_attacks
        end if
    end for
    if new_attack then
        attack_group = array
        push anomaly to attack_group
        push attack_group to possible_attacks
    end if
end for
```

Whenever a packet receives an anomaly score larger than 0, an attempt is made to group the packet towards similar anomalous packets, thus forming the basis for a potential attack. This is done by comparing the source and destination addresses of the packet toward the attacks that have already been
detected. If there is a match, the packet is added to the previous attack – but only if it occurred within a one hour time span of the other packets in the attack. If not, it is added as a new attack. This is also the case if the anomalous packet does not match any of the previously detected attacks.

The next section explains how Inspectobot has been implemented, before the Test Programme is presented in Section 4.1.
3.7 Implementation

GRIDAC has been implemented in Inspectobot using the Python programming language, and consists of three main components, also written using Python. These are:

- *inspecto-generate*, which is responsible for generating the rules.
- *inspecto-filter*, which compares unknown traffic towards the rule hierarchy, calculates anomaly scores, and attempts to group packets according to different requirements.
- *inspectoweb*, which is the graphical frontend to the output from both inspecto-generate and inspecto-filter, and allows an analyst to look into the data that has been produced.

**inspecto-generate**

The component *inspecto-generate* represents the "Training" mode of Inspectobot, as illustrated in Figure 3.5. It is responsible for generating multiple rules that are organized in a hierarchical manner, based on certain input parameters from the user, as specified in **REQ1**. The required parameters are divided into two main categories, and shown in Tables 3.6 and 3.7.

<table>
<thead>
<tr>
<th>Name</th>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of bytes</td>
<td>-b</td>
<td>Number of bytes to consider, starting with the IP header. Default is 48 bytes.</td>
</tr>
<tr>
<td>Bits per group</td>
<td>-g</td>
<td>The number of bits each feature in the rule should represent. The default is 8 (implying 8 bits in 1 byte).</td>
</tr>
<tr>
<td>Training packets</td>
<td>-p</td>
<td>The amount of (randomly selected) packets that will be loaded from the input file.</td>
</tr>
<tr>
<td>Artificial</td>
<td>-a</td>
<td>If set, the input file is treated as an artificial dataset.</td>
</tr>
<tr>
<td>Randomize</td>
<td>-r</td>
<td>If set, objects from the input file are loaded in random order.</td>
</tr>
</tbody>
</table>

The above table shows the parameters that are required for defining how the input file should be parsed. This is a very important in terms of customizing the various experiments that will be performed – as it enables the possibility of setting the amounts of bits per group, the amount of bytes that should be inspected for each packet, and also makes it possible to use artificial datasets.

Table 3.7, as shown below, lists the required parameters that are used during the training phase.
<table>
<thead>
<tr>
<th>Name</th>
<th>Switch</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target ratio</td>
<td>-t</td>
<td>The ratio of packets the generated rule should match. (e.g. 50%)</td>
</tr>
<tr>
<td>States per action</td>
<td>-s</td>
<td>The number of states per action for each TA.</td>
</tr>
<tr>
<td>Force converge</td>
<td>-c</td>
<td>If the TA is unable to decide between two actions, it is forced to make a decision based on its current state, after a given amount of iterations. (e.g. 10000).</td>
</tr>
<tr>
<td>Tree depth limit</td>
<td>-d</td>
<td>Maximum hierarchy depth.</td>
</tr>
<tr>
<td>Node limit</td>
<td>-l</td>
<td>Stop expanding the hierarchy if one of the leaf nodes classifies the given ratio of the total input file.</td>
</tr>
</tbody>
</table>

These parameters are equally important as those listed in Table 3.6. Target ratio enables the user to set the target classification ratio of a given rule. Note that the actual classification ratio might differ somewhat from the target ratio, as it is merely used as a guideline for the teams of TA. The two last parameters are used to minimize the possibility of creating rule hierarchies that are too strict – meaning that there could be a single rule per packet. Node limit effectively eliminates this problem by checking that a rule does not classify more than a given ratio of the total dataset.

To better understand what a rule might look like after it has passed through inspecto-generate, consider the following example. If the number of bytes (-b) is set to 48 bytes, and the amount of bits per group (-g) is set to 8, the following rule might be created once all the TA have converged using the format $f_n = \alpha_x(u))$.

```
C(01000101) C(00000000) C(00000010) I(01011101) I(10100011) I(01110000)
C(01000000) C(00000000) C(01111110) C(00000110) C(00000000) C(00000000)
C(10111100) C(01111110) C(11001000) C(00001111) C(00001010) C(00000000)
C(00001010) I(00110101) I(10010011) I(01011010) C(00000000) C(00010000)
I(10000000) I(01111100) I(00100000) I(10010100) I(00001001) I(11100011)
I(01011101) I(11110111) C(10000000) C(00011100) C(00000000) I(00111010)
I(00100110) I(11100100) C(00000000) C(00000000) C(00000001) C(00000001)
C(00001000) C(00000000) C(00000000) C(00110100) I(01111000) I(01110110)
```

Figure 3.6: Example rule with bit grouping set to 8, and the number of inspected bytes to set to 48.

The rule is read from left to right, and from top to bottom. The first rule element has been set to constant, denoted by the letter C, and describes the first eight bits in the IP header. Note the bitstring 01000101 in the first element. The first four bits specifies which IP protocol in use*, and the last 4

* Unless there are other network protocols involved, like IPv6, the first four bits are always constant, and set to 0100 (or 4 in decimal notation).
bits specifies the amount of 32-bit words\(^*\) in the IP header.

Using *graphviz\(^†\)*, it is also possible to automatically create a graphical visualization of how the generated rules relate to each other, as shown in Figure 3.7. The purpose of the below figure is not to show the details of the rule tree, but to present its overall complexity.

The above figure displays a rule tree with a depth of 13 levels. The number of levels is determined by the number of rows in the tree. Table 3.8 provides an explanation of the different symbols found in Figure 3.7.

![Graphical representation of a rule hierarchy, created automatically using graphviz.](image)

Figure 3.7: Graphical representation of a rule hierarchy, created automatically using graphviz.

Table 3.8: Rule Tree Symbol Explanation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Symbol" /></td>
<td>Input dataset that is used in the training process.</td>
</tr>
<tr>
<td><img src="image" alt="Symbol" /></td>
<td>Node describing more traffic than the specified node limit. Unless the tree depth level has been reached, this node will be expanded into another tree level.</td>
</tr>
<tr>
<td><img src="image" alt="Symbol" /></td>
<td>Node describing less traffic than the specified node limit. It will not be expanded further.</td>
</tr>
</tbody>
</table>

Note that the node limit in the tree depicted in Figure 3.7 was set to 5%. No maximum depth was specified, which allowed the tree to expand until the node limit was reached in all branches of the tree.

\(^*\) As shown in Figure 2.4 on page 25, the amount of 32-bit words in the IPv4 header is set to 0101 (or 5 in decimal notation), unless any additional options are set.  
\(^†\) Graphviz is open source graph visualization software. [http://www.graphviz.org/](http://www.graphviz.org/)
Inspectobot’s three components have one thing in common. They all rely on a database to both store and retrieve information. When inspecto-generate has finished generating a hierarchy, it uses a Python module called pickle* for serializing (and de-serializing) the object structure as a physical file that can be stored in a file system. The path to the pickle object is then saved in a database, as displayed by the Entity Relationship Model in Figure 3.8, along with all the parameters that were used in generating the hierarchy. In the below figure, each attempt to create a rule hierarchy is referred to as a ‘Run’.

![ER diagram of Inspectobot’s Database Structure](http://docs.python.org/library/pickle.html)

Figure 3.8: ER diagram of Inspectobot’s Database Structure

For each run, a series of tests can be performed. This is handled by inspecto-filter, and the results are stored in the tables TestRun, Anomaly, AnomalyFields and Attack.

There are two parameters required when using inspecto-filter. The first is the pathname to the dataset containing the unknown packets, and the second is the location of the pickle object that contains the rule hierarchy used when classifying packets.

* http://docs.python.org/library/pickle.html
When *inspecto-filter* is initialized, it matches every packet in the input dataset, and attempts to classify them as normal or anomalous.

If an anomalous packet is detected, *inspecto-filter* looks up information about the relevant nodes in the rule hierarchy, and temporarily caches this information. Then, it attempts to categorize the anomalous packets into different attacks, as shown in Algorithm 3.10 on page 44. Each possible attack is also cross-referenced with the aforementioned Detection Truth Lists included in the DARPA IDS Evaluation Sets.

Finally, all the results are stored in the database, and it can then be retrieved using the graphical user interface, *inspectoweb*.

**inspectoweb**

The graphical user interface, *inspectoweb*, makes it easier for an analyst to view the anomalous packets, and get a detailed overview of current events and possible attacks. It is written as a web application using Python and Javascript – and utilizes the web development frameworks, Pylons* and jQuery†, in addition to the Python object relational mapper (ORM) SQLAlchemy‡.

![Screenshot of Inspectoweb](image)

**Figure 3.9:** Screenshot of Inspectoweb, the Graphical User Interface to Inspectobot. It presents a list of grouped events to the user.

The screenshot displayed in Figure 3.9 shows a list of events (or anomalies) that are grouped by the source and destination addresses. The anomalous packets are collected from the DARPA IDS set. Additional information generated by *inspecto-generate* and *inspecto-filter* is also displayed.

It is also possible to get a detailed view of all the possible attacks, where the analyst has the opportunity to filter packets based on attack identification numbers, as shown in Figure 3.10.

Figure 3.10: This view gives the analyst a list of possible attacks, with the possibility of filtering them based on their identification number.

This enables the possibility of filtering all the packets that are related to a possible attack, such that the packets can be further analyzed in tools like Wireshark*.  

* http://www.wireshark.org
Chapter 4

Testing and Validation

In this chapter, verification of GRIDAC and the prototype Inspectobot is carried out. The research questions that were defined in Section 1.2 on page 5 are also researched, and the findings are presented here.
4.1 Test Programme

In order to thoroughly investigate the research questions, a formal Test Programme was created. This test programme included a set of test cases, carefully designed to evaluate different aspects of Inspectobot. The test programme is displayed in Table 4.1.

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC 1</td>
<td>GRIDAC Parameter Tuning</td>
<td>Determine the optimal parameters for use with artificial data and real life network packets, such as the amount of states per action, target ratio and force converge limit.</td>
</tr>
<tr>
<td>TC 2</td>
<td>Classification Evaluation with Artificial Data</td>
<td>Determine that Inspectobot is able to mine frequent itemsets from a dataset. Generate several datasets using a collection of pre-defined template objects. Inspectobot will then be set to classify these datasets, and the purpose in each case is to identify patterns matching the pre-defined template objects.</td>
</tr>
<tr>
<td>TC 3</td>
<td>IDS Evaluation with Artificial Data</td>
<td>Determine that Inspectobot is able to function as an IDS. Using two artificially generated datasets - one normal and one containing anomalies - see if Inspectobot is able to filter out objects that do not match the rules generated when training on the normal data set.</td>
</tr>
<tr>
<td>TC 4</td>
<td>Classification Evaluation with Network Packets</td>
<td>Determine that Inspectobot is able to extract frequent patterns from a dataset consisting of real network packets. Using the 1999 DARPA IDS Evaluation Set, see if Inspectobot is able to generate rules that can be used to classify different network packets.</td>
</tr>
<tr>
<td>TC 5</td>
<td>IDS Evaluation with Network Packets</td>
<td>Determine that Inspectobot can be used to detect anomalies in network packets. Using the 1999 DARPA IDS Evaluation Sets, investigate if Inspectobot is able to detect the listed attacks.</td>
</tr>
</tbody>
</table>

Of these test cases, TC 5 was divided into two subtests, which are presented in Table 4.2.
Table 4.2: Test Case 5 - Subtests

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC 5.1</td>
<td>Attacks Detected</td>
<td>How many of the attacks are actually detected by Inspectobot, and which attacks remain undetectable.</td>
</tr>
<tr>
<td>TC 5.2</td>
<td>Detection Rate</td>
<td>Based on the amount of detected attacks, how many of these are real positives compared to false positives?</td>
</tr>
</tbody>
</table>

The details of the different tests were documented in separate test case documents, and in Appendix B, an example is included. The key findings that were gathered during the execution of the test cases are included in the following sections.

4.2 GRIDAC Parameter Tuning

The underlying parameters to Inspectobot can be tuned and altered in a variety of ways. In order to find the optimal combination of all the parameters, several preliminary tests were conducted. The details of these tests are not included here, but the derived parameters are presented.

In Table 3.6 on page 46, a brief explanation of the different parameters was given. Two sets of parameter values were derived from the preliminary tests; one for testing with artificial datasets, and another for testing with network packets.

Table 4.3: Parameter Values used in Test Programme

<table>
<thead>
<tr>
<th>Name</th>
<th>Switch</th>
<th>Artificial Data</th>
<th>Network Packets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of bytes</td>
<td>-b</td>
<td>10</td>
<td>48</td>
</tr>
<tr>
<td>Bits per group</td>
<td>-g</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Training packets</td>
<td>-p</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>Artificial</td>
<td>-a</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>Randomize</td>
<td>-r</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Target ratio</td>
<td>-t</td>
<td>90 %</td>
<td>90 %</td>
</tr>
<tr>
<td>States per action</td>
<td>-s</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>Force converge</td>
<td>-c</td>
<td>1000</td>
<td>10000</td>
</tr>
<tr>
<td>Tree depth limit</td>
<td>-d</td>
<td>Not set</td>
<td>Not set</td>
</tr>
<tr>
<td>Node limit</td>
<td>-l</td>
<td>Varies</td>
<td>Varies</td>
</tr>
</tbody>
</table>

Table 4.3 describes the parameter values that were used in all of the test cases described in the Test Programme. As can be seen, there are some differences between the parameters used for artificial datasets and those used for network packets.
The reason for these differences is that the artificial datasets are of a smaller size than the datasets containing network packets. Therefore, a larger number of bytes are needed when dealing the larger data sets (like the DARPA IDS Evaluation Sets). Also, since a certain number of packets are selected randomly from the entire dataset during the training process, more packets are needed in order to get an accurate representation of a larger dataset. More packets available in the training process also means that the force converge parameter can be set to a higher value. This is to allow for more exploration.

For both artificial and network packet datasets, the tree depth limit was disabled in order to allow the tree to grow to its full potential, reaching the set node limit in all branches before halting tree growth.

One of the more influential parameters of Inspectobot is the parameter controlling the number of states per action used in the training process. More states per action leads to increased accuracy, but also causes performance to drop, as each LA will require more iterations in order to converge.

To illustrate how the number of states per action affects the tree generation performance, Figure 4.1 shows how the total number of iterations required to generate a tree - using the same input data in all cases - increases as a direct consequence of adjusting the number of states per action.

![Figure 4.1: The number of required iterations needed to generate a tree in relation to the number of states per action that is used.](image)

By looking at the curve in Figure 4.1, it becomes evident that the number of states per action has a significant effect on the number of iterations that are required in order to generate a tree. After several tests, the number of states per action parameter was set to a value of 70. This provided a good balance between accuracy and performance.

After the input parameters had been determined, the rest of the test cases could be executed.
4.3 Classification Evaluation with Artificial Data

Determining all the frequent itemsets in a dataset consisting of network packets is a very complex task, and almost impossible for a human entity to accomplish. Because of this, it is hard to verify that all patterns have actually been correctly identified. To avoid this problem, several simple, artificially generated datasets with known patterns was used. Using artificial datasets, the classification aspect of Inspectobot could be verified.

Dataset Description

A number of different objects, generated from template objects which were defined in dataset templates, were included in the artificial datasets. Using only these datasets, the goal of each test was to correctly identify the recurring patterns that were defined when the dataset was first generated. By doing this, Inspectobot would be able to mine all the objects that were generated from one template object - matching the designed dataset split ratios perfectly.

A brief description of the different datasets that were generated is given in Table 4.4.

Table 4.4: Dataset Overview

<table>
<thead>
<tr>
<th>ID</th>
<th>Size</th>
<th>Split Ratios</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DS 1</td>
<td>100000 objects</td>
<td>25 %, 50 %, 75 %, 100 %</td>
<td>Consists of four different objects, where some of the objects are subsets of other objects.</td>
</tr>
<tr>
<td>DS 2</td>
<td>100000 objects</td>
<td>12.5 %, 25 %, 37.5 %, 50 %,</td>
<td>The same as DS 1, but with eight objects instead of four.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>62.5 %, 75 %, 87.5 %, 100 %</td>
<td></td>
</tr>
<tr>
<td>DS 3</td>
<td>100000 objects</td>
<td>4 x 25 %</td>
<td>Consists of four objects, where the feature values of the objects are completely unique. The only split ratio that can be achieved is 25 %.</td>
</tr>
<tr>
<td>DS 4</td>
<td>100000 objects</td>
<td>8 x 12.5 %</td>
<td>The same as DS 3, but with eight objects instead of four.</td>
</tr>
</tbody>
</table>

Each dataset provided Inspectobot with a unique challenge. DS 1 challenged Inspectobot’s ability to distinguish between objects that had some feature values in common. The different object types that were added to this dataset can be seen in Figure 4.2.

* Split ratio refers to the total percentage one object type represents in the dataset. Achieving a split ratio of 0.2, a rule that matches 20 % of the objects in the dataset would have to be created.
DS 1 contains 25000 objects of each object type, adding up to 100000 objects in total. Because Inspectobot has been set to group 8 bits together, the resulting rule should consist of four rule elements. For instance, in order to classify 75% of the objects present in DS 1, the rule described in Figure 4.3 would have to be generated.

\[
\text{C(11111111) C(11111111) I(00000000) I(00000000)}
\]

Figure 4.3: Rule classifying 75% of the objects in DS 1.

By varying the number of constant fields, Inspectobot had to match all four possible split ratios. DS 2, which was designed in the same way as DS 1, had eight possible split ratios.

In addition to testing if Inspectobot was able to identify objects which contained similarities, it was also necessary to see how it would manage when the template objects were 100% different from one another. To test this, DS 3 and DS 4 were created. Figure 4.4 provides an example of four of the objects contained in these datasets.

\[
\begin{align*}
100000000100000001000000010000000000000000 \\
01000000100000000100000001000000010000000000000000 \\
00100000010000000010000000010000000010000000000000000 \\
0001000000010000000010000000001000000000000000010000000000000000
\end{align*}
\]

Figure 4.4: Examples of objects present in DS 3 and DS 4. One object per line.

When grouping 8 bits together, these objects do not have any feature values in common. Therefore, it is impossible to create rules that describe more than one object type at a time.
Artificial Classification Results

Using this dataset collection, the classification aspects of Inspectobot could be evaluated. Table 4.5 shows the results from the tests, where 5 iterations per dataset were used.

Table 4.5: Results from testing the classification aspect of Inspectobot

<table>
<thead>
<tr>
<th>Identified Template Objects</th>
<th>DS 1</th>
<th>DS 2</th>
<th>DS 3</th>
<th>DS 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4/4</td>
<td>8/8</td>
<td>4/4</td>
<td>8/8</td>
</tr>
<tr>
<td>2</td>
<td>4/4</td>
<td>8/8</td>
<td>4/4</td>
<td>8/8</td>
</tr>
<tr>
<td>3</td>
<td>4/4</td>
<td>8/8</td>
<td>4/4</td>
<td>8/8</td>
</tr>
<tr>
<td>4</td>
<td>4/4</td>
<td>8/8</td>
<td>4/4</td>
<td>8/8</td>
</tr>
<tr>
<td>5</td>
<td>4/4</td>
<td>8/8</td>
<td>4/4</td>
<td>8/8</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>4/4</td>
<td>8/8</td>
<td>4/4</td>
<td>8/8</td>
</tr>
</tbody>
</table>

Table 4.5 shows that all template objects were correctly identified, meaning that Inspectobot had successfully learned all of the frequent itemsets present in all of the datasets.

Now that the classification aspect of Inspectobot had been verified, the next step was to determine if it would function as an IDS - separating unknown objects from normal objects by matching against learned frequent itemsets.

4.4 IDS Evaluation with Artificial Data

In order to determine if Inspectobot was able to detect and filter out anomalies, two artificial datasets were generated; one dataset containing normal objects that was used for training, and one dataset containing both normal and anomalous objects that was used for testing.

An anomalous object is an object that does not support any of the learned frequent itemsets in a tree. For each object that is filtered through a given tree, an anomaly score is calculated. If the score is positive, Inspectobot sees the object as an anomaly. If the anomaly score is negative, it is considered normal, and no alarms are raised.

Evaluation Method

The dataset that was used for training, contained one million objects which were based on seven different template objects. Figure 4.5 gives an overview of the different objects that were included in the dataset, in addition to the
amount of each object type that was included. A * indicates that the value of that particular object feature can be either 0 or 1 for that object type.

<table>
<thead>
<tr>
<th>Percentage</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
<th>Value 5</th>
<th>Value 6</th>
<th>Value 7</th>
<th>Value 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>30%</td>
<td>0</td>
<td>*</td>
<td>1</td>
<td>*</td>
<td>1</td>
<td>*</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30%</td>
<td>0</td>
<td>1</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10%</td>
<td>0</td>
<td>*</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12%</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>8%</td>
<td>0</td>
<td>0</td>
<td>*</td>
<td>0</td>
<td>1</td>
<td>*</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>5%</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5%</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 4.5: The proportion of different object types that are found in the artificial dataset used for training Inspectobot.

When Inspectobot was trained on this dataset, the bits per group parameter was set to a value of 1. This meant that each value in Figure 4.5 was considered an object feature value, and resulted in 8 groups per object.

After Inspectobot had created rules to describe the frequent patterns in the dataset, the resulting tree was tested against a dataset which contained anomalous objects. In addition to the normal object types described in Figure 4.5, this dataset also included a variable amount of three additional objects, which differed from the normal ones in varying degrees. Figure 4.6 contains an overview of these objects.

<table>
<thead>
<tr>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
<th>Value 5</th>
<th>Value 6</th>
<th>Value 7</th>
<th>Value 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 4.6: The proportion of different anomalous object types that are found in the artificial dataset used for testing Inspectobot.

The objects in Figure 4.6 were grouped to form four different artificial attacks, and were then mixed in with the normal objects to emulate background noise.

* The values for these fields are selected randomly when the datasets are generated.
Artificial IDS Results

As previously mentioned, the objects in Figure 4.6 should be assigned higher anomaly scores than the normal objects when they are compared to the rules in the tree. Figure 4.7 shows the anomaly score distribution in the artificial dataset after running the test dataset through the generated tree.

Figure 4.7: The anomaly score distribution for an artificial dataset containing four artificial attack instances. The peaks indicate the locations of the attacks.

The anomaly distribution displayed in Figure 4.7 clearly shows that there are four areas which have higher anomaly scores than the rest of the dataset. These areas correspond to the four artificial attacks that were inserted into the dataset. This shows that Inspectobot was able to detect the anomalous objects, and that they were assigned a higher anomaly score.

As can also be seen in the figure, one attack instance generated a lower anomaly score than the others - the reason being that the objects in this attack had more features in common with the normal objects.

4.5 Classification Evaluation with Network Packets

Having determined that Inspectobot was able to mine frequent itemsets and that it could detect anomalies in artificially generated datasets, the next step was to determine how well it would perform with datasets containing real network packets. To determine this, a customized network packet dataset and the 1999 DARPA IDS Evaluation Sets were taken into use.

The 1999 DARPA IDS Evaluation Data Sets

In 1999, DARPA contracted the Lincoln Laboratory at MIT to develop a range of datasets with the purpose of testing the performance of existing IDS. The datasets try to simulate network traffic occurring in a small US Air Force base
that is connected to the Internet. The internal network is connected to the Internet by a CISCO router, and network traffic is captured on both sides of this router. The captured traffic is made available in separate files for inside and outside network traffic.

Only the network packets captured on the outside of the CISCO router was used when evaluating Inspectobot, as explained in the Key Assumptions and Limitations on page 19.

A total of five weeks worth of network traffic is provided in the set; Three weeks for training, and two weeks for testing. Week 1 and 3 are attack free, while week 2 contains labeled attacks. Week 4 and 5 contains over 200 labeled attacks in total. [18]

**Split Criteria Analysis**

Before evaluating Inspectobot with the DARPA dataset, Inspectobot’s ability to generate a single rule from a dataset consisting of network packets needed to be investigated. To accomplish this, a custom made dataset was used. This dataset was created manually, and consisted of traffic towards two different IP addresses, hosting a combined total of two services.

Exactly the same amount of network packets for both IP addresses was included, where both hosts were represented by 500 network packets each, adding up to 1000 network packets in total.

The dataset was filtered to only include network packets in one direction. Only traffic directed towards one of the two servers was included - everything else was removed. This was done to reduce the randomness of the values contained in the IP destination address field, making it easier to interpret the results.

The motivation for filtering the dataset in this way was to determine if the prototype could identify and use the IP destination address field as a split criterion, and thereby achieve a perfect split ratio of 50%.

Due to how GRIDAC works, a classification target of 50% means that the rewards for selecting either a constant field or a wildcard field is equal. This leaves the randomly selected filter packet and the randomly drawn training packets with the deciding factor on how the generated rule turns out*.

---

* If an automaton is unable to decide whether to choose a constant or a wildcard for its object feature, a manually adjustable limit can be used to force the automaton to decide after a given number of iterations. This keeps the training process from stagnating.
Figure 4.8 shows the inner workings of a rule that was generated using the manually created dataset, along with the achieved split ratio.

As shown in the figure, only one constant field is active in the classification process, matching a classification percentage of 49.9%, which is very close to the target of 50%.

Although the outcome deviated from what was expected, Inspectobot completed its task successfully, classifying 49.9% of the objects in the dataset.

To understand why the system did not manage to classify exactly 50% of the dataset, the generated rule, displayed in Figure 4.9, had to be analyzed.

---

* Even though only one constant field is active in the classification process, there may be other constant fields present in the rule as well, but they could be shadowed by an earlier occurring constant field, rendering them inactive.

---

Figure 4.9: A rule generated with a classification target of 50%, showing the different split criteria that has been used.

The third rule element in the rule, which has been highlighted in the above figure, is the IP Length field. The four other highlighted rule elements correspond
to the four octets that together make up the *IP destination address*.

As can be seen in the figure, the *IP address destination fields* were in fact identified as split criteria, but all four of them were shadowed by the earlier occurring *IP Length* field. That is, the *IP Length* split criterion had already classified all of the network packets that would have been taken care of by the four *IP destination address* rule elements.

Despite of this, a near perfect classification percentage was achieved, classifying 0.01 % less than the optimum.

Having determined that Inspectobot could successfully identify object split criteria in a dataset containing network packets, the next step was to see if it would be able to generate a tree describing 100 % of a given week in the 1999 DARPA IDS Evaluation Set.

**Tree Generation**

To achieve this, Inspectobot was set to examine one full week of training data. Once finished, the resulting rules were examined. An example of the size of a tree generated from one week of network packets can be seen in Figure 4.10.

![Figure 4.10: Example of a tree that was generated with one week of training data from the 1999 DARPA IDS Evaluation Set. Each square corresponds to one rule.](image)

As can be seen, multiple rules were needed in order to classify the entire dataset. In addition, a lower node limit of 5 % was used, meaning that the generated tree could have become even more detailed if permitted. A description of the different symbols in the Figure can be reviewed in Table 3.8 on page 48.
To gain further insight into the tree generation process, Figure 4.11 provides an example of how many iterations the LA required to generate this tree. Since the 48 first bytes of each network packet is inspected, 48 object features - each with one dedicated LA - are active in the training process.

![Figure 4.11: The number of iterations that were required for each object feature to generate a tree with a 5% node limit.](image)

Figure 4.11 gives an indication of which object features Inspectobot struggled the most with. Low iteration values indicate that it was easy for Inspectobot to decide on an action for that particular object feature. Higher values indicate that it had trouble deciding.

When Inspectobot has trouble deciding on an action for a given feature, the struggling automata will be forcefully converged - one by one - in order to prevent stagnation. As previously explained, this process is done sequentially, and that explains why the number of iterations for some of the automata gradually increases along with the object feature number in the figure.

To ensure that Inspectobot had been able to successfully learn the normal traffic patterns in the dataset, the finished tree was tested against the same dataset that was used in the training process. This yielded no anomalous objects, so the conclusion that Inspectobot was able to detect and learn normal traffic patterns could be drawn.

### 4.6 IDS Evaluation with Network Packets

Now that Inspectobot had proven itself able to fully describe a fairly large dataset containing real network packets, the next step was to determine if it could separate abnormal network packets from normal network packets, and flag them as anomalies as necessary.
Evaluation Method

When evaluating the detection rate of an IDS, there are three important factors to consider:

- The amount of false positives generated; Normal traffic falsely reported as anomalous by the IDS.
- The amount of real positives generated; Attacks correctly identified by the IDS.
- The amount of false negatives generated; Attacks not identified by the IDS.

To determine if an attack was real or not, detailed knowledge of the attacks present in the DARPA set was required. Thankfully, DARPA provides records of the attacks in what they refer to as the *identification and scoring truth*. All detected anomalies were cross-referenced with these records in order to measure the rate of false positives, real positives, and false negatives.

Figure 4.12 describes how the IDS detection performance for Inspectobot was measured.

![Figure 4.12: Basic process flow for the attack evaluator used when comparing detected attacks against the DARPA identification and scoring truth.](image)

As Figure 4.12 shows, Inspectobot first reads the network packets, filters out any anomalies, groups the anomalies into several attacks, before the attacks
are compared to the DARPA identification and scoring truth records. Here, the real positive, false positive, and false negative rates are calculated and used to generate a ROC curve, which shows the overall detection performance to Inspectobot for the total number of attacks contained in the DARPA set.

Several trees were generated during the test. The tree generation parameters remained constant throughout the entire test case - with the exception of the parameter controlling the node limit and the input file used for training. Three different node limit values were tested in order to document the effect each value had on the detection rate. In addition, different combinations of the training data from week 1 and 3 were tested, in order to determine if some combinations would achieve better detection rates than others. For all combinations of the training data and the tree generation parameters, three iterations were executed in order to determine the consistency of the results.

**Attack Detection**

To get an impression of how Inspectobot is able to distinguish between normal and anomalous traffic, some examples of frequent itemsets - or *rules* - and which attacks they can identify have been included in Table 4.6.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Attacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ver+ihl:0x45, frag1:0x40, frag2:0x00, proto:0x06, srcport1:0x00, tcphl:0x50, urgptr1:0x00, urgptr2:0x00</td>
<td>ps</td>
</tr>
<tr>
<td>ver+ihl:0x45, dscp:0x00, frag1:0x00, frag2:0x00, proto:0x06, tcphl:0x50, urgptr1:0x00, urgptr2:0x00</td>
<td>ps</td>
</tr>
<tr>
<td>ver+ihl:0x45, dscp:0x00, len:0x00, frag1:0x00, frag2:0x00, ttl:0x40, proto:0x06, dstaddr1:0xac, dstaddr2:0x10, dstport1:0x00, dstport2:0x17, tcphl:0x50, recwd1:0x7d, recwd2:0x78, urgptr1:0x00, urgptr2:0x00, pld1:0x00, pld5:0x00, pld7:0x00, pld8:0x00, pld9:0x00</td>
<td>ps, guesstelnet, sendmail</td>
</tr>
</tbody>
</table>

As seen, each rule consists of selected bytes from a packet, combined with a hexadecimal representation of the corresponding byte value. Thus, considering the first row of the table, network packets of the so-called ps-attack do not match the frequent itemset ver+ihl:0x45, frag1:0x40, frag2:0x00, proto:0x06, proto:0x06,
srcport1:0x00, tcphl:0x50, urgptr1:0x00, urgptr2:0x00, and are therefore reported as anomalies.

**IDS Performance**

When evaluating Inspectobot’s IDS performance, only the outside data from the DARPA set was used. All of the generated trees were tested against week 4 and 5 of the DARPA set, and the results are shown in the form of several ROC (Receiver Operator Characteristic) curves. As explained in Figure 4.12, the rate of real positives, false positives and false negatives is calculated for all of the generated trees. These rates are used as a basis for the ROC curves, where the rate of real positives runs along the $y$-axis, and the rate of false positives runs along the $x$-axis.

For each real positive that is detected, a point is added to the curve. The $y$-coordinate denotes the current rate of real positives, while the $x$-coordinate denotes the current rate of false positives that have been detected so far. A false positive rate of 1 means that 100% of the false positives that were generated during the test had occurred when $x$ true positives had occurred.

For example, the point $(0.5, 0.3)$ denotes that when 30% of all attacks contained in the dataset had been identified, 50% of all generated false positives had at that time occurred.

ROC curves are very useful for visualizing how the generated trees perform in different areas of the dataset. They will often show different attacks that are detected - at various timestamps in the dataset. A curve that quickly moves towards $(1, 1)$ indicates that the tree in question has a high detection rate, and that it has generated a low amount of false positives.

The detection rate achieved by each tree, is determined by the highest $y$-value in the ROC charts. The amount of false positives can be seen as points along the $x$-axis.
As previously mentioned, three node limits were tested for each dataset. Three trees were generated for each node limit, and in Figure 4.13, the best trees from each node limit are displayed. One ROC chart per training dataset has been included below.

Figure 4.13: ROC charts showing how the IDS performance of Inspectobot varies depending on the training data and node limit that is used.

As illustrated in Figure 4.13, there were some variations in the results. The trees that got the best detection rates were the ones that were generated with a node limit of 5% - and particularly the ones trained with only week 1.
The amount of false positives changed in accordance with the set node limit. A node limit of 5 % provided for a moderate amount of false positives. The trees generated with a node limit of 1 % had a lower detection rate, and also showed an increase in the amount of false positives generated. Finally, the trees generated with a 10 % node limit generated less false positives than in the other cases, but were also the ones with the poorest detection rate.

To further demonstrate the effect the node limit had on the detection sensitivity, Figure 4.14 shows the anomaly scores for three trees that were generated with different node limits. The same input data was used in all cases.

Figure 4.14: Comparison of anomaly distribution between three trees generated with different node limits.

69
As displayed in Figure 4.14, the majority of the dataset received anomaly scores below 0. Knowing that a typical computer network contains more normal traffic than anomalous traffic, this was in line with what was expected. A spike in the anomaly score indicates that an anomaly is present. Notice that as the node limit increases, the number of spikes decreases. Although the number of detected anomalies is at its highest when a low node limit has been set, all of these anomalies are not necessarily considered real positives.

To give a more comprehensive view of Inspectobot’s detection capabilities, Table 4.7 provides an overview of the results that were gathered from all of the tests that were executed.

Table 4.7: Results from IDS evaluation using the DARPA set

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Ratio</th>
<th>False Positives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Data</td>
<td>Node Limit</td>
<td>avg</td>
</tr>
<tr>
<td>Week 1 Outside</td>
<td>1 %</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td>5 %</td>
<td>0.73</td>
</tr>
<tr>
<td></td>
<td>10 %</td>
<td>0.43</td>
</tr>
<tr>
<td>Week 3 Outside</td>
<td>1 %</td>
<td>0.54</td>
</tr>
<tr>
<td></td>
<td>5 %</td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td>10 %</td>
<td>0.54</td>
</tr>
<tr>
<td>Week 1+3 Outside</td>
<td>1 %</td>
<td>0.54</td>
</tr>
<tr>
<td></td>
<td>5 %</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td>10 %</td>
<td>0.46</td>
</tr>
</tbody>
</table>

Table 4.7 shows that the best detection rates were achieved by the trees that were trained on week 1, and used a node limit of 5 %.

To get an impression of the overall detection capabilities of Inspectobot, all of the results from all of the generated trees have been combined, and in Table 4.8, an overview of the attack types that remain undetected by all trees is given.

Table 4.8: Undetected DARPA Attack Types

<table>
<thead>
<tr>
<th>Alias</th>
<th>Instances</th>
<th>Console/remote</th>
<th>Inside/outside</th>
<th>Stealthy</th>
</tr>
</thead>
<tbody>
<tr>
<td>anypw</td>
<td>2</td>
<td>console</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>dict</td>
<td>6</td>
<td>remote</td>
<td>inside</td>
<td>no</td>
</tr>
<tr>
<td>guesspop</td>
<td>1</td>
<td>remote</td>
<td>outside</td>
<td>no</td>
</tr>
<tr>
<td>illegalsniffer</td>
<td>14</td>
<td>remote</td>
<td>inside</td>
<td>mix</td>
</tr>
<tr>
<td>land</td>
<td>2</td>
<td>remote</td>
<td>both</td>
<td>no</td>
</tr>
<tr>
<td>ntfsdos</td>
<td>3</td>
<td>console</td>
<td>-</td>
<td>no</td>
</tr>
<tr>
<td>resetscan</td>
<td>1</td>
<td>remote</td>
<td>inside</td>
<td>yes</td>
</tr>
<tr>
<td>sshprocesstable</td>
<td>12</td>
<td>remote</td>
<td>inside</td>
<td>no</td>
</tr>
</tbody>
</table>

Given that only the outside data from the DARPA set was used, the attacks that occur on the inside were undetectable by Inspectobot. Also undetectable
are the locally executed attacks that do not generate any network traffic. This leaves two detectable attacks that were not detected by Inspectobot: one instance of *guesspop* and one instance of *land* (the other instance is on the inside).
Chapter 5

Discussion

This chapter discusses the results that were gathered during the execution of the Test Programme, and links these findings to the research questions, formed in Section 1.2.

5.1 Frequent Pattern Mining in Network Packets

The work presented in the past chapters introduces a scheme known as GRIDAC, and a prototype named Inspectobot. Using teams of learning automata for mining frequent patterns in network packets (or artificially generated datasets), RQ 1 asked if it was possible to generate rules for modelling the normal behavior. To investigate this theory, test case TC 1, in addition to TC 2 and TC 4 in particular, were defined.

The purpose of TC 1 was to determine the optimal parameters required to achieve the best possible results such that the remaining test cases could be carried out.

Parameter Tuning

During this evaluation, several changes were done to GRIDAC, including adding support for Markov Chains, as described in Section 3.4. By making it possible to move several states in each iteration, the number of iterations a TA needed in order to converge was significantly reduced. Another addition was the ability to forcefully converge a given TA. This was done in order to prevent the scheme from stalling when it encountered difficulties selecting an action.

Moreover, it was discovered that a variable bit grouping could lead to a negative impact on the overall classification process. If a feature contains eight bits (like a byte in a network packet), Inspectobot should be configured to group the features on exactly eight bits - and not four, two or one. The preliminary results showed that a more appropriate bit grouping lead to better classification of
data, and that the exact opposite happened in other cases. When the bits per
group parameter was set to one (and feature objects containing 8 bits were
used), the TA were more likely to converge towards constant, where wildcard
would have been the better choice.

Also, in some of the first preliminary tests that GRIDAC was put through,
it became apparent that the rules, in some cases, were too strict to be useful
at the lower levels in the tree. To combat this, support for specifying a lower
node limit was added. (An option for specifying the tree depth limit was also
added, but as the node limit was added, this parameter became deprecated.)
In effect, adding the node limit parameter stopped tree growth in areas that
described less than its set value.

Setting the node limit can however be challenging. The preliminary tests
showed that if it was set too low, some of the generated rules became too strict,
which lead to a higher false positive rate while evaluating the IDS aspects.
Setting the node limit too high would cause the rules to become too general to
be used for IDS purposes. The false negative rate would increase as a result.
These aspects are discussed further page 76.

When the parameter optimization process was completed, and Inspectobot
was properly tuned, REQ 1 had been satisfied and TC2 could be started, in
which artificial datasets were generated for testing the classification aspects of
Inspectobot.

Classification with Artificial Datasets

With four pre-generated artificial datasets, carefully designed to evaluate and
verify the classification process it would be easier to determine if Inspectobot
was behaving properly - as defined in TC 3. When this was not the case,
meaning that the results were not in accordance with the expectations, figuring
out the reason behind the faulty results was usually a trivial matter. As real-
world network packets are quite complex, and contain more data, it would be
more difficult to locate and correct any errors.

Each of these datasets contained specific template objects, and the ultimate
goal while evaluating them, was to identify their recurring patterns, such that
rules could be generated. The final outcome of this test case was an overall
success. Inspectobot was able to generate rules for identifying the template
objects, and an example is defined as "C₀ ∗ C₁ ∗ C₀ C₀ C₁ ∗". Some of
these datasets were also created to verify that hierarchical structuring of the
rules were possible. As such, both REQ 3 and REQ 4 were satisfied, and its
classification aspects (when dealing with artificial datasets) had been verified.

The next step was to see if Inspectobot behaved correspondingly while classi-
fying network packets.
Classification with Network Packets

In order to verify that Inspectobot was able to mine frequent itemsets in real-world network packets, as stated in TC 4, two different datasets were used during the evaluation process. In addition to the 1999 DARPA IDS Evaluation Sets, a customized dataset consisting of manually generated network traffic towards two unique IP addresses was created, where each host acted as a server that hosted a specific service, and where the number of packets were evenly divided among the hosts. The reason for including a customized dataset of network packets was to evaluate if Inspectobot was able to achieve a perfect split (of 50 %) in the dataset based on the IP destination addresses.

As opposed to classification with artificial datasets, Inspectobot was now configured to inspect the 48 first bytes in each network packet, and with the bit grouping parameter set to eight.

With the manually generated network packets, Inspectobot was indeed able to generate a rule for dividing the packets, as shown in Figure 4.8. Although the result was not perfect, it was highly satisfactory, as the generated rule was able to represent 49.9 % of the network packets by using the IP Length header field as split criterion. This indicated that 49.9 % of all the packets in the dataset had this particular field set to a specific value, while the remaining 50.1 % of the packets would be represented by slightly different split criteria. This indicates that the choice of filter packet (i.e. the randomly selected packet which is used as base for creating a rule) had an impact on the overall result when only a single rule was generated.

In addition to the customized dataset, Inspectobot was set to examine one full week of training data from the DARPA set, and as explained in Section 4.5, Inspectobot was able to create a tree structure containing multiple rules. As such, REQ 2 was satisfied. When this process was completed, it also became apparent that the choice of filter packet was not as essential as when a single rule was created. Since multiple rules are generated for each level, multiple filter packets are drawn, and the hierarchy continues to grow regardless of the order they are being drawn.

To ensure that Inspectobot had been able to successfully learn the normal traffic patterns in the dataset, the finished tree was tested against the same dataset that was used in the training process. This yielded no anomalous objects, so the conclusion that Inspectobot was able to detect and learn normal traffic patterns could be drawn.

Impact of Anomaly Scoring

The implemented anomaly scoring functionality worked surprisingly well. This was first noted during TC 2, where Inspectobot was set to classify unknown data in an artificially generated dataset. As shown in Figure 4.7, most of the data was classified as "normal", with a few exceptions that resulted in anomaly
score peaks. Not all of these peaks were reported as anomalous however, since
the anomaly threshold was set to 0. In this particular case, lowering the
anomaly score threshold to $-2$ would have detected the final anomaly as well,
without introducing any additional false positives.

However, the same cannot be said for the anomaly distribution graphs pre-
sented in Figure 4.14. Lowering the anomaly score threshold in these cases
might have lead to more real positives being detected, but it would also sig-
ificantly increase the number of false positives.

Nonetheless, the anomaly scoring worked according to its purpose, and it can
be concluded that there is a tradeoff between the number of detected real
positives and false positives, and that the anomaly score threshold can be
modified to adjust it.

**Impact of Node Limit**

Based on the results from evaluating Inspectobot with the DARPA IDS Eval-
uation Sets, as presented in Table 4.7, it became evident that the node limit
affected the detection rate and the amount of false positives generated.

The rule hierarchies that have been generated with a node limit of 5 % appear
to provide the best detection rate. Those that were generated with a node limit
of 1 % did not provide a higher detection rate, but contributed to a significant
increase in false positives. In contrast, the rule hierarchies with a node limit of
10 % provided, in all cases, a lower detection rate compared to both the other
hierarchies. It also returned a much lower amount of false positives.

The amount of false positives in rule hierarchies with a node limit of 1 %
suggest that the network packets, which in other cases would be classified as
normal, are not able to traverse the rule hierarchy far enough to achieve an
anomaly score below 0, and thus be classified as such.

With rule hierarchies that have a node limit of 10 %, the generated rules
appear to be more general - meaning that packets which in other cases would
be classified as anomalous, is wrongly being classified as normal.

Based on the hierarchies that used the node limit of 5 %, week 1 reported an
average detection rate of 73 %, while week 3 achieved an average detection
rate of 63 %. With both weeks combined, however, an average detection rate
of 61 % was reported. These results are likely explained by the increase in
the amount of network packets considering both weeks, as it might raise the
probability of creating generic rules, especially if the there are large differences
between the packets in week 1 and 3.

For this reason, estimating the optimal node limit appears to be a difficult task.
To achieve the best detection rate, it seems necessary to generate several rule
hierarchies with a variable node limit, and determine which is better, based
on the amount of false positives generated.
Concluding remarks concerning RQ 1

By evaluating Inspectobot using both artificial datasets, and real-world network packets, it was proven that the prototype was, in fact, able to detect the underlying semantics in the datasets. With this information in hand, the normal traffic patterns could be modeled using a tree structure of rules.

5.2 Potential as an Intrusion Detection System

With NETAD, Mahoney [8] was able to achieve good results with the 1999 DARPA IDS Evaluation Sets - by analyzing the first 48 bytes in IP packets, where it was able to detect 132 of 185 attacks, with 100 false alarms. Because of NETADs notable detection rate at 71.5%, a similar approach was applied to GRIDAC, and implemented in Inspectobot - as it also inspected the first 48 bytes in IP packets. However, unlike NETAD, in which nine pre-generated models were used for detecting the attacks, Inspectobot remained completely unsupervised, and it was also able to classify data going in both directions (from WAN to LAN and vice versa).

As RQ 2 asks how good GRIDAC (and its prototype implementation, Inspectobot), is at detecting anomalies compared to an existing solution, TC 5 was defined.

Empirical Results

TC 5 stated that Inspectobot should be evaluated with the 1999 DARPA IDS Evaluation Sets. By training on one full week of attack-free data (week 1), Inspectobot was set to classify another week of data (week 4) containing attacks – and was at best able to detect 51 out of 62 possible attacks, as shown in Appendix B – giving a total detection rate of 82%, with 56 false alarms. When the same training data was used to classify both weeks containing attacks (week 4 + week 5), it managed to achieve an average detection rate of 73% with 123 false alarms, as displayed in Table 4.7. This rate is unfortunately not representable for the remaining tests that were performed. Using the same node limit as in the previous results, 63% of the attacks were detected when Inspectobot was trained with Week 3 (instead of Week 1), and 61% of the attacks when both Week 1 and Week 3 were used for training.

Determining the amount of attacks in the 1999 DARPA IDS Evaluation Sets

The 1999 DARPA Detection Scoring Truth *, and Identification Scoring Truth † lists were used for cross-referencing the anomalies reported by Inspectobot

† http://www.ll.mit.edu/mission/communications/ist/files/master_identifications.list
in order to determine the detection rate. According to [18], these lists contain over 200 instances of 58 attack types, which are distributed over two weeks. The attacks had also been categorized depending on whether they were console or network based. Also, the network based attacks were categorized as ”inside” or ”outside”, stating which side of a firewall the attacks occurred. Based on calculations presented in Figure 5.1, 148 attacks were defined as being outside of the firewall, while 40 were defined as being inside. In addition, 12 of the attacks were defined as console based, or network based (but without any additional information about which side of the firewall they had occurred). These numbers were not listed in [18], but were gathered from the aforementioned Detection Scoring Truth list using the POSIX commands listed in Figure 5.1.

```
# Lists the amount of total attack incidents for both weeks.
$ egrep "^ID" master_identifications.list | wc -l
  201

# Lists the amount of outside attacks for both weeks.
$ egrep "^\[0-9][0-9]\..*\sout\s" master-listfile-condensed.txt \
  | cut -c1-10 | sort -u | wc -l
   148

# Lists the amount of inside attacks for both weeks.
$ egrep "^\[0-9][0-9]\..*\sin\s" master-listfile-condensed.txt \
  | cut -c1-10 | sort -u | wc -l
   41

# Lists amount of console based attacks, and remote attacks that
# are listed as neither "inside" nor "outside", for both weeks.
$ egrep '^[0-9]\..*\{8,\}(auto|man).*(rem|cons)\s' \n  master-listfile-condensed.txt | cut -c1-10 | sort -u | wc -l
   12
```

Figure 5.1: Amount of real attacks in the DARPA IDS Evaluation Set

In his paper on NETAD [8], Mahoney states that there are 185 ”inside” attacks in the 1999 DARPA IDS Evaluation Sets, of which 132 were detected. Based on the fact that [18] list the amount of attacks as more than 200 (they do not give an exact amount), and that the number of ”inside” attacks, as shown in Figure 5.1, are only 41, it could be possible that Mahoney uses a different way of counting the amount of attacks on the inside. Nevertheless, Inspectobot relies on the data provided by the truth lists that are provided alongside the 1999 DARPA IDS Evaluation Sets. For this reason alone, no further investigation has been done in regards to determining why these numbers are different.

It should however be mentioned that the list in question had not been formatted in a way that made manual lookup of potential attacks trivial. For this particular reason, lookup scripts had to be created manually in order to automate this process, so that the timestamps of the attacks detected by Inspectobot could be compared against the truth list automatically. Performing
the lookup using scripts leaves out the human factor, which could lead to faulty results.

In any way, the results from evaluating Inspectobot with the 1999 DARPA IDS Evaluation Sets are still quite conclusive. The prototype is able to detect anomalous packets, and it is also able to group together these packets, based on the involved IP addresses and their timestamps - such that possible attacks can be detected.

**Undetected Attacks**

According to the test results, not all of the attacks were detected. The reason for this problem is, however, quite simple. Inspectobot attempts to model the normal behavior in the network packets that are used for training. If any attacks correspond to "normal" behavior to such a degree that the anomaly score falls below zero, it will not be flagged as anomalous. Lowering the bar for flagging a potential attack as anomalous could lead to more real attacks being identified, but this would also generate more false alarms.

Two distinct examples are given in Table 4.8 on page 70, where the attacks land and guesspop remained undetectable. According to the attack database shipped alongside the 1999 DARPA IDS Evaluation Sets, the Land attack is a denial-of-service attack that is effective against some older TCP/IP implementations, where a spoofed SYN packet is sent to the vulnerable system. The attack is also recognizable because the IP source and destination fields are identical (which should never exist on a properly working network). It is likely that this particular attack would be easier to detect by a rule based IDS, as it is a trivial task to create a rule that triggers an alarm whenever it sees a network packet with identical source and destination IP addresses.

Guesspop, on the other hand, appears to be a standard dictionary based attack towards a server running POP (Post Office Protocol). Upon closer inspection of this particular attack however, it appears that the attack was not successful, as the aforementioned 1999 DARPA IDS Truth lists reports. Using the protocol analyzer Wireshark, it was discovered that the attack was performed by a client using the IP address 172.16.112.194 towards the POP daemon running on 202.247.224.89. During the 30 attempts, the attacker tried to log in to the server with the username alie, and a password combination that consisted of the letters alie, in combination with a number sequence ranging from 0 to 29. For this reason, Inspectobot was in fact correct when it reported the attack as "normal", since people often enter the wrong usernames and/or passwords when they try to log in to Internet services. However, it still remains unclear if the attack would have been detected, had it been successful.

**Concluding remarks concerning RQ 2**

RQ 2 asks how good Inspectobot is at detecting anomalies, compared to an existing solution, with regards to the amount of real positives versus the amount
of false positives detected. Since Inspectobot cannot be directly compared to Mahoney’s results, the detection rate can instead be used as an indicator. On average, Inspectobot was able to detect 73% of the “outside” attacks in Week 4 and Week 5, when Week 1 had been used for training the prototype. Meanwhile, NETAD reported a detection rate of 71.5%, although [8] claimed to use the “inside” network traffic as opposite to “outside”, which Inspectobot relied on.

Inspectobot was also able to detect an error in the 1999 DARPA IDS Evaluation Sets. The error was in fact an attack that was reported as successful in the official truth lists, when it upon closer inspection was in fact unsuccessful.

The next section discusses the final research question that was stated.
5.3 Possible replacement for Snort?

To answer RQ 3, Inspectobot is currently not a valid replacement for Snort, nor will it probably ever be. A-NIDS in general, is still a valuable asset to any security analyst, and should be used alongside Snort, such that novel attacks can be detected, but also for finding underlying network problems. As mentioned in Section 1.3.5 on page 12, Snort did not perform well on 1998 DARPA IDS Evaluation Sets, as the datasets only contain a limited number of attacks that are detectable by signatures. Inspectobot was on the other hand able to detect many of them.

The main drawback with Inspectobot, is the amount of processing capacity that is required for running it optimally. The current pros and cons with the prototype are listed in Table 5.1.

<table>
<thead>
<tr>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Able to detect anomalous packets and group them as attacks.</td>
<td>Not optimized for reading packets from network interfaces. Although it is supported, it does lead to large amounts of dropped packets in its current state.</td>
</tr>
<tr>
<td>No human supervision is required for generating the rules, or for classifying unknown packets</td>
<td>Requires a great deal of processing power, in addition to available memory and disk space.</td>
</tr>
<tr>
<td>GUI makes it easy for an analyst to get an overview of the various attacks</td>
<td>The prototype does not support editing or modifying the generated rules. However, expanding the hierarchy with additional rules is possible.</td>
</tr>
<tr>
<td>Good detection rate (82 %) of attacks in one week of the 1999 DARPA IDS Evaluation Set, with an average detection rate of 73 % in both weeks.</td>
<td></td>
</tr>
</tbody>
</table>

Snort is usually listening on a network interface in order to detect attacks in live streams of network traffic. This is also possible to do with Inspectobot, although it will result in large amount of packets being dropped by the operating system kernel. The reason for this is that the classification (and matching) process is quite CPU intensive. As such, Inspectbot, in its current state, is not able to keep up with the incoming and outgoing packet rate, and is thus not capable of being used actively on a network interface.

For this reason, the prototype should have been implemented in a programming language like C right from the start to reduce the amount of system calls needed to do the required operations. However, this would also have made modifications to both GRIDAC and Inspectobot much more difficult, as it would require more code, and likely several redesigns of the algorithms.
lot of time, programming and debugging wise, was saved by using a high-
level object oriented language like Python, and the end result is a working
prototype. Although it is somewhat inefficient and probably bloated code
wise, it still suits it purpose, and it also has its own Graphical User Interface,
successfully satisfying REQ 5.

5.4 Remarks concerning the 1999 DARPA IDS Evaluation Sets

The 1999 DARPA IDS Evaluation Sets contain several minor errors, which
have made processing and analysis of the attacks a time consuming process.
As previously mentioned, two attack truth lists were provided alongside the
datasets, where one of them listed when the various attacks took place, while
the other listed the amount of unique attacks.

During a cross-reference of these lists, typographic errors were detected in both,
and especially with regards to the attack names – making it much harder to
estimate the amount of unique attacks. As an example, the list containing
the unique attacks, mentions both "xterm" and "xterm1", although it appears
that this is in fact the same attack.

The lists in question were also very unstructured, making it a tedious process
to perform automatic lookup of the attacks reported by Inspectobot. The
truth lists were quite detailed, and contained a lot of information – but they
were not formatted in a standard way, like CSV (comma separated values).
To give an example, the only way to look up if an attack was in fact real or
not, was to cross-reference with the following information:

- When the attack started (although the timestamps were not given in
  UTC).
- Which IP addresses that were involved.

It would have been much better if the packet numbers were listed in addition to
the above information, as this would have simplified the lookup process greatly.
If the lists had also been formatted using SQL, that would have simplified the
process even further.
Chapter 6

Conclusion

The work presented in this thesis has demonstrated that the Grimstad Data Classifier (GRIDAC) scheme is able to model normal behavior in complex data formats like network packets. For this reason, GRIDAC was implemented as an Anomaly Based Network Intrusion Detection System (A-NIDS) called Inspectobot, which was evaluated using the 1999 DARPA IDS Evaluation Set. Inspectobot was powered by a team of hierarchically structured Learning Automata (LA) that have the unique property of operating in unknown environments. Due to their low computational complexity, they were well suited for the task in question.

6.1 Empirical Results

Inspectobot, like any network anomaly detector, does not attempt to describe the nature of an attack, nor does it try to determine if an event is hostile or not. Instead, it attempts to find unusual or interesting patterns in a vast amount of data, tag them as anomalous - and bring them to the attention of a security analyst for further investigation.

In extensive evaluation using both artificial data and data from the 1999 DARPA IDS Evaluation Sets, the results are quite conclusive - demonstrating that the prototype shows an excellent ability to find frequent itemsets, such that a large set of network packets can be modeled in the form of hierarchically structured rules. Furthermore, the sets of frequent itemsets produced for network intrusion detection are compact, yet accurately describe the different types of network traffic present, making it possible to detect attacks in the form of anomalies.

By training on one full week of attack-free data in the DARPA IDS Evaluation Sets, Inspectobot was at best able to detect 51 out of 62 possible attacks when it was configured to classify a second week of data containing attacks. Thus, the detection rate in that particular case was 82 %, and it also reported 56 false alarms. When Inspectobot was set to classify both weeks containing attacks,
using the same week as in the previous case for training, it managed to achieve an average detection rate of 73 %, with 123 false positives.

Also, Inspectobot was able to detect an error in the DARPA sets, where an attack was wrongfully listed as successful, when it in fact was not.

6.2 Conclusions and Implications

The main goal of the work presented in this thesis has been reached, as Inspectobot is a fully working A-NIDS that is able to mine frequent itemsets, and detect anomalous patterns in network packets using teams of hierarchically structured Learning Automata.

During the evaluation of Inspectobot, using artificial data and real-world network packets, it was also discovered that it was able to detect anomalous patterns which can be regarded as attacks. Although the prototype was unable to detect some of the attacks in the DARPA IDS Evaluation Sets, it was still able to achieve a surprisingly good detection rate – considering the fact that it performed the classification process completely unsupervised, and with no previous knowledge of the attacks in question.

In its current state, Inspectobot is not usable outside of a testbed environment, and in order to obtain satisfactory results from evaluations, the input data, which is used for both the training and classification processes, needs to be collected in advance.

Inspectobot is currently not a replacement for Snort, the de-facto rule based Intrusion Detection System, but it is very likely able to serve alongside Snort if implemented in a more efficient programming language like C or C++.

6.3 Future Work

To increase Inspectobot’s efficiency, which is currently its largest drawback, it should be rewritten in a high level programming language with less abstraction* than Python, such as C or C++, which generate far more efficient code. The first priority is to rewrite the component responsible for classifying unknown data as either normal or anomalous, as this needs to be as fast as possible in order to cope with larger packet rates.

It is less important to prioritize the component responsible for mining the frequent itemsets, and generate hierarchically structured rules, as this is more difficult to accomplish - mainly due to the complete rewrite of all the algorithms in use.

The next version of Inspectobot should make it possible to tune the generated rule hierarchies, without having to restart the process from scratch. As an

* In this context, abstraction refers to Python's use of high-level data types, modularity and use of dynamic typing.
example, if a new host is connected to an existing network in which Inspectobot has already learned the normal traffic patterns, the traffic originating to and from the new host might be classified as anomalous.

Using a Bayesian approach, it might also be possible to increase the amount of real positives, and reduce the amount of false positives. If a human analyst goes through the anomalies reported by Inspectobot, and labels these as positive or negative, this data could be used for better classification of unknown data. This might be possible by identifying those nodes in a given tree that provide the best detection rate, and make a decision based on that information.
Bibliography


Appendix A - Work Package Example

The following work package was defined in Table 3.1 on page 29. As seen on the next page, it is wrapped in an informative front page that explains its purpose, the expected amount of time for completion, and the actual time used.
# Work Package

**Inspectobot Graphical User Interface**

<table>
<thead>
<tr>
<th>Inspectobot Graphical User Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Purpose</strong></td>
</tr>
<tr>
<td><strong>Participants</strong></td>
</tr>
<tr>
<td><strong>Requirement(s)</strong></td>
</tr>
<tr>
<td><strong>Estimated Time Usage</strong></td>
</tr>
<tr>
<td><strong>Actual Time Usage</strong></td>
</tr>
</tbody>
</table>
Purpose

One of the key parts of an Intrusion Detection System is that a security analyst is able to interact with it through an interface. Inspectobot can be administered from a command line interface, but from the user’s perspective, the graphical user interface (GUI) will make the system easier to use on a higher lever, and also simplifies the process of gaining an overall view of the available events that need attention.

Choosing a Framework

The purpose of a framework is to improve the efficiency of creating new software. By allowing the programmer to spend more time on meeting software requirements and developing algorithms, frameworks can help improve productivity in addition to raising the quality, reliability and robustness of the software in question.

Since the other components of Inspectobot is written in Python, it is also reasonable to write the GUI using the same language. Django and Pylons are two of the most popular web frameworks that use Python. Both of the frameworks are mature and tested in production in some major websites, with Pylons being used by the user-driven news-site Reddit.

According to a discussion on the pros and cons of either web framework, Django seems to be the obvious choice for blogs and newspaper sites, as one of its main requirements during development was to make entire sites quickly, such as blogs and newspaper sites. It also appears to be more ‘user friendly’ to developers unfamiliar with Python. This isn’t because Pylons is a lot harder to learn than Django, so much as it is that Pylons’s biggest advantage lies in its ease of customization. But to really use that customizability, the developer needs to be more aware of what python software is available.

Dusko Jordanovski has tried to write a non-biased comparison between Django and Pylons, and states that Django has more magic and less code, while pylons has more code and less magic. He also writes that Pylons is essentially a bare-bones wrapper around the WSGI specification that uses 3rd party modules for templating, database interaction, routing and just about anything else, while Django is aimed towards rapid development of web applications and has everything packed inside of it - it’s own template system, routing and object relational mapper (ORM). Apparantly, this allows Django to establish high reusability for code between different projects. On the other hand, the developer is limited to one ORM and templating system.

Being that Inspectobot GUI will not contain features found in most web sites today, the ability for customization is essential. Since the developers are also

\* http://www.djangoproject.com/
\† http://pylonsproject.org/
\‡ http://en.wikipedia.org/wiki/Reddit
\§ http://j.mp/CjH55
\¶ http://jordanovski.com/django-vs-pylons
familiar with Python in general, Pylons seems to be the better choice. For this reason, Pylons is selected as the web framework.

Design

The design should be simple and attempt to adhere to the KISS∗ principle, which implies that simplicity should be a key goal in design, and that unnecessary complexity should be avoided.

The most important information should be available to the user from one screen, such as an:

Executive summary that provides general statistics over all the detected anomalies, the amounts of test runs that being analyzed in addition to information related to the anomaly score for all events.

Analyst View that provides the necessary detailed information about the packets that have been flagged as anomalies. This includes the source and destination IP addresses, TCP/UDP ports, anomaly score and similar. However, the information should not be more detailed than necessary. In case the analyst needs to inspect a group of events in more detail, this should be done in a different view.

Attack Summary which provides the analyst with a summary of all the detected attacks. This includes the amounts of events related to the attack and also the name of the attack should it be available. The naming aspect will currently only be available when the DARPA set is being analyzed.

It should also be possible to manage the different rule hierarchies that have been generated, as well as choosing which test runs that should be analyzed.

Use of Object Relational Mappers and Code Examples

Wikipedia defines Object Relational Mapping as a programming technique for converting data between incompatible type systems in object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language.†

Inspectobot currently uses the relational database MySQL as backend for information storage. At an earlier time, however, it used the more minimalistic variant SQlite, but this was later changed due to processing efficiency. Luckily, SQLAlchemy had been implemented as an object relational mapper (ORM)

from the start, so switching from SQLite to MySQL proved to be a rather trivial problem.

An ORM essentially allows you to access a database using objects from within the programming language. As an example, the SQL query

```
SELECT users.username from users WHERE users.id = 1
```

would be the equivalent of

```
user = ORM.query(User.username).filter(User.id == 1).
```

The ORM that is recommended to use with Pylons is called SQLAlchemy∗ which, according to the SQLAlchemy website, provides a full suite of well known enterprise-level persistence. Use of Object Relational Mappers and Code Examples patterns, designed for efficient and high-performing database access, adapted into a simple and Pythonic domain language.

The following code example shows how a table in the database is being mapped to an object, using the SQLAlchemy declarative syntax.

```python
class Anomaly(Base):
    
    """
    This table contains a list of all the anomalies that have been detected in a given test run.
    """
    __tablename__ = 'anomaly'
    id = Column(Integer, primary_key = True)
    testrun_id = Column(Integer, ForeignKey('testrun.id'), primary_key=True)
    attack_id = Column(Integer, ForeignKey('attack.id'), default=0)
    destination_address = Column(String(255))
    destination_nationality = Column(String(255), default="Unknown")
    source_address = Column(String(255))
    source_nationality = Column(String(255), default="Unknown")
    destination_port = Column(String(255))
    source_port = Column(String(255))
    timestamp = Column(DateTime(timezone=True))
    anomaly_score = Column(Integer)
    parent_node_id = Column(String(255))
    anomalyfields = relation(
        'AnomalyFields',
        backref='anomaly',
        primaryjoin="AnomalyFields.anomaly_id==Anomaly.id",
        cascade='all'
    )
```

Figure WP4.1: Code example that defines the mapping between SQLAlchemy and the anomaly table in the database.

As seen in the above code, SQLAlchemy is told which columns that are used as primary and foreign keys. A relation between the tables Anomaly and AnomalyFields is defined, where cascade has been set to all. This implies that

∗ http://www.sqlalchemy.org/
whenever a row gets updated or deleted from Anomaly, the related rows in AnomalyFields will be deleted as well.

GUI in use

The following screenshot displays how the Event view looks like. This is similar to the Analyst View defined in the Design section.

Figure WP4.2: Screenshot of Inspectoweb, the Graphical User Interface to Inspectobot. It presents a list of grouped events to the user.

In this particular view, the events are grouped based on the source and destination IP addresses, and an executive summary is given. Thus, the analyst is provided a table that contains the following information:

- **First seen**, which specifies when the first packet (of those that are grouped) was detected.
- **Events**, which provides a counter of the total number of anomalous packets between the IP addresses in question.
- **Source Port**
- **Source Address** (with geographic information, if possible)
- **Destination Address** (with geographic information, if possible)
- **Destination Port**
- **Duration**, which calculates the duration between the first and last of the grouped packets.
- **Parent Node ID**, which lists the parent rule in the hierarchy which classified the packets as anomalous.
• **Max anomaly score**, which lists the maximum anomaly score for all the packets that are grouped together.

It is also possible to get a detailed view of all the possible attacks, where the analyst have the opportunity to filter packets based on attack identification numbers, as shown in Figure WP4.3.

Figure WP4.3: This view gives the analyst a list of possible attacks, with the possibility of filtering them based on their identification number.

This enables the possibility of filtering all the packets that are related to a possible attack, such that the packets can be further analyzed in tools like Wireshark*.

In order to select which test runs that should be analyzed, the following view is given:

Figure WP4.4: This view gives the analyst a list of runs (rule hierarchies) and corresponding test runs.

A **run** is the equivalent of a rule hierarchy, and the table of hierarchies (or trees) is denoted as **Forest** in the above figure. A single run can be associated with several test runs.

* [http://www.wireshark.org](http://www.wireshark.org)
Shortcomings

At the moment, the GUI can only be used for displaying anomalous packets and possible attacks. It is also possible to list the build parameters for the rule hierarchies and to display a graphical representation of them.

Due to limited amounts of time, there are some shortcomings in the GUI which should be fixed before the project is released into the public domain. At the moment, it is currently not possible to:

- Start the rule hierarchy build process from the GUI
- Create test runs from the GUI
- Store the anomalous packets in the pcap format for further analysis with Wireshark

These features are available from a CLI environment however, so adding them to the GUI should not pose much difficulty.

Summary

This Work Package have presented a summary of the Inspecto GUI and the frameworks involved with developing it. It currently meets the initial design specifications, but before an eventual public release, the listed shortcomings should be fixed.

The time spent on designing and programming the GUI was approximated to 150 hours, while the actual time usage slightly exceeded 170 hours.

April 13th, 2010

Vegard Haugland
Appendix B - Test Case

Example

The test cases provided in this appendix were defined in Table 4.1 on page 53. Each test case is wrapped in an informative front page that explains its purpose, the expected amount of time for completion, and the actual time used.
## Inspectobot Test Case

### IDS Evaluation with DARPA IDS Set

<table>
<thead>
<tr>
<th>Description</th>
<th>Using the 1999 DARPA IDS Evaluation Sets, it is necessary to investigate if Inspectobot is able to detect the listed attacks, and how many false positives that are generated</th>
</tr>
</thead>
</table>
| Participants                                                                 | Vegard Haugland  
Marius Kjølleberg  
Svein-Erik Larsen                                                                                                                                         |
| Estimated Time Usage                                                         | 4 weeks                                                                                                                                                    |
| Actual Time Usage                                                            | 6 weeks                                                                                                                                                    |
Purpose

In Section 1.2 on page 5, a set of research questions was formed. RQ 2, asked how good GRIDAC is at detecting anomalies compared to an existing solution and to what extent false positives and false negatives are generated. RQ 3 then asked whether the A-NIDS implementation of GRIDAC is able to replace the current R-NIDS implementation like Snort.

The purpose of this test case is to investigate how GRIDAC compares to these research questions.

Test Setup and Tree Generation

To evaluate the IDS performance of Inspectobot, the 1999 DARPA IDS Evaluation Set was used for both training and testing purposes. Despite being dated, the set is still considered a viable choice for testing anomaly based IDS. A total of five weeks worth of network traffic is provided in the set; Three weeks for training, and two weeks for testing. Week 1 and 3 are attack free, while week 2 contains labeled attacks. Week 4 and 5 contains over 200 labeled attacks.

As explained in Section 1.5, time limitations dictate that only the attacks categorized as ”outside” will be analyzed. From this a list of tests have been composed.

<table>
<thead>
<tr>
<th>TC5.1 TC5.1: Test Setup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training</td>
</tr>
<tr>
<td>Week 1</td>
</tr>
<tr>
<td>Week 3</td>
</tr>
<tr>
<td>Week 1+3</td>
</tr>
</tbody>
</table>

As described in Table TC5.1, Week 1, Week 3 and Week 1+3 will be used for training the system, while Week 4 and Week 5 will be used for testing. Each instance of training will consist of three runs with different node limit (1%, 5% and 10%) and each node limit will be run three times to ensure that the system is consistent. Parameters described in 3.6 on page 46 will be used for training.

The 1999 DARPA Detection Scoring Truth*, and Identification Scoring Truth† lists will be used for locating the anomalies in the Evaluation Set in order to determine how many anomalies are detected and calculate the detection rate.

---

† [http://www.ll.mit.edu/mission/communications/ist/files/master_identifications.list](http://www.ll.mit.edu/mission/communications/ist/files/master_identifications.list)
By using the POSIX commands listed in Figure TC5.1 it is concluded that there are 148 attacks possible to detect when using the "Outside" Evaluation Set. Of these 148 attacks, 62 are found in Week 4 and 86 in Week 5.

```bash
# Lists the amount of outside attacks for both weeks.
$ egrep "^\[0-9]\[0-9\]..\s\sout\s" master-listfile-condensed.txt \ | cut -c1-10 | sort -u | wc -l
148

# Lists the amount of outside attacks for Week 5
$ egrep '^[0-9]\[0-9\]..\s04/0[5-9]\..\sout\s' \ master-listfile-condensed.txt | cut -c1-10 | sort | uniq \ | wc -l
86
```

Figure TC5.1: Amount of real attacks in the DARPA IDS Evaluation Set

Table TC5.2 describes the expressions used the results table.

<table>
<thead>
<tr>
<th>ID</th>
<th>Unique Identification test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run</td>
<td>Run number in specific test</td>
</tr>
<tr>
<td>Limit</td>
<td>Node Limit used in specific test</td>
</tr>
<tr>
<td>Testing</td>
<td>Training file</td>
</tr>
<tr>
<td>Training</td>
<td>Testing file(s)</td>
</tr>
<tr>
<td>TA</td>
<td>Total attacks in testing file(s)</td>
</tr>
<tr>
<td>RP</td>
<td>Total attacks detected during test</td>
</tr>
<tr>
<td>FP</td>
<td>Total false positive detected during test</td>
</tr>
<tr>
<td>Rate</td>
<td>Detection rate</td>
</tr>
</tbody>
</table>
### Results

#### Week 1

TC5.3 TC5.3: Week 1 - Outside - Node Limit 1%

<table>
<thead>
<tr>
<th>ID</th>
<th>Run</th>
<th>Limit</th>
<th>Training</th>
<th>Testing</th>
<th>TA</th>
<th>RP</th>
<th>FP</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Week 1 Outside</td>
<td>Week 4 Outside Week 5 Outside</td>
<td>62</td>
<td>43</td>
<td>199</td>
<td>0.69</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>Week 1 Outside</td>
<td>Week 4 Outside Week 5 Outside</td>
<td>62</td>
<td>43</td>
<td>179</td>
<td>0.69</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>Week 1 Outside</td>
<td>Week 4 Outside Week 5 Outside</td>
<td>62</td>
<td>43</td>
<td>176</td>
<td>0.69</td>
</tr>
</tbody>
</table>

TC5.4 TC5.4: Week 1 - Outside - Node Limit 5%

<table>
<thead>
<tr>
<th>ID</th>
<th>Run</th>
<th>Limit</th>
<th>Training</th>
<th>Testing</th>
<th>TA</th>
<th>RP</th>
<th>FP</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>Week 1 Outside</td>
<td>Week 4 Outside Week 5 Outside</td>
<td>62</td>
<td>49</td>
<td>48</td>
<td>0.79</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>5</td>
<td>Week 1 Outside</td>
<td>Week 4 Outside Week 5 Outside</td>
<td>62</td>
<td>50</td>
<td>48</td>
<td>0.80</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5</td>
<td>Week 1 Outside</td>
<td>Week 4 Outside Week 5 Outside</td>
<td>62</td>
<td>51</td>
<td>56</td>
<td>0.82</td>
</tr>
</tbody>
</table>

TC5.5 TC5.5: Week 1 - Outside - Node Limit 10%

<table>
<thead>
<tr>
<th>ID</th>
<th>Run</th>
<th>Limit</th>
<th>Training</th>
<th>Testing</th>
<th>TA</th>
<th>RP</th>
<th>FP</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>1</td>
<td>10</td>
<td>Week 1 Outside</td>
<td>Week 4 Outside Week 5 Outside</td>
<td>62</td>
<td>31</td>
<td>26</td>
<td>0.50</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>10</td>
<td>Week 1 Outside</td>
<td>Week 4 Outside Week 5 Outside</td>
<td>62</td>
<td>31</td>
<td>18</td>
<td>0.50</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>10</td>
<td>Week 1 Outside</td>
<td>Week 4 Outside Week 5 Outside</td>
<td>62</td>
<td>31</td>
<td>26</td>
<td>0.50</td>
</tr>
</tbody>
</table>
### TC5.6 TC5.6: Week 3 - Outside - Node Limit 1%

<table>
<thead>
<tr>
<th>ID</th>
<th>Run</th>
<th>Limit</th>
<th>Training</th>
<th>Testing</th>
<th>TA</th>
<th>RP</th>
<th>FP</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>1</td>
<td>5</td>
<td>Week 3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>41</td>
<td>193</td>
<td>0.66</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>45</td>
<td>236</td>
<td>0.52</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>5</td>
<td>Week 3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>34</td>
<td>185</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>40</td>
<td>241</td>
<td>0.47</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>5</td>
<td>Week 3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>42</td>
<td>188</td>
<td>0.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>38</td>
<td>238</td>
<td>0.44</td>
</tr>
</tbody>
</table>

### TC5.7 TC5.7: Week 3 - Outside - Node Limit 1%

<table>
<thead>
<tr>
<th>ID</th>
<th>Run</th>
<th>Limit</th>
<th>Training</th>
<th>Testing</th>
<th>TA</th>
<th>RP</th>
<th>FP</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1</td>
<td>10</td>
<td>Week 3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>38</td>
<td>33</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>47</td>
<td>63</td>
<td>0.55</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>10</td>
<td>Week 3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>46</td>
<td>31</td>
<td>0.74</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>51</td>
<td>50</td>
<td>0.59</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>10</td>
<td>Week 3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>45</td>
<td>37</td>
<td>0.72</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>51</td>
<td>51</td>
<td>0.59</td>
</tr>
</tbody>
</table>

### TC5.8 TC5.8: Week 3 - Outside - Node Limit 10%

<table>
<thead>
<tr>
<th>ID</th>
<th>Run</th>
<th>Limit</th>
<th>Training</th>
<th>Testing</th>
<th>TA</th>
<th>RP</th>
<th>FP</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
<td>Week 3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>35</td>
<td>23</td>
<td>0.56</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>44</td>
<td>37</td>
<td>0.51</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>1</td>
<td>Week 3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>37</td>
<td>22</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>43</td>
<td>36</td>
<td>0.50</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>1</td>
<td>Week 3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>37</td>
<td>24</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>45</td>
<td>37</td>
<td>0.52</td>
</tr>
</tbody>
</table>
## TC5.9 TC5.9: Week 1+3 - Outside - Node Limit 1%

<table>
<thead>
<tr>
<th>ID</th>
<th>Run</th>
<th>Limit</th>
<th>Training</th>
<th>Testing</th>
<th>TA</th>
<th>RP</th>
<th>FP</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>1</td>
<td>1</td>
<td>Week 1+3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>40</td>
<td>178</td>
<td>0.65</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>43</td>
<td>236</td>
<td>0.50</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>1</td>
<td>Week 1+3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>39</td>
<td>190</td>
<td>0.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>39</td>
<td>246</td>
<td>0.45</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>1</td>
<td>Week 1+3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>37</td>
<td>179</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>40</td>
<td>225</td>
<td>0.47</td>
</tr>
</tbody>
</table>

## TC5.10 TC5.10: Week 1+3 - Outside - Node Limit 5%

<table>
<thead>
<tr>
<th>ID</th>
<th>Run</th>
<th>Limit</th>
<th>Training</th>
<th>Testing</th>
<th>TA</th>
<th>RP</th>
<th>FP</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>1</td>
<td>5</td>
<td>Week 1+3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>43</td>
<td>31</td>
<td>0.69</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>55</td>
<td>63</td>
<td>0.64</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>5</td>
<td>Week 1+3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>38</td>
<td>30</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>50</td>
<td>54</td>
<td>0.58</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>5</td>
<td>Week 1+3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>36</td>
<td>27</td>
<td>0.58</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>49</td>
<td>41</td>
<td>0.57</td>
</tr>
</tbody>
</table>

## TC5.11 TC5.11: Week 1+3 - Outside - Node Limit 10%

<table>
<thead>
<tr>
<th>ID</th>
<th>Run</th>
<th>Limit</th>
<th>Training</th>
<th>Testing</th>
<th>TA</th>
<th>RP</th>
<th>FP</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>1</td>
<td>10</td>
<td>Week 1+3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>37</td>
<td>21</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>41</td>
<td>40</td>
<td>0.48</td>
</tr>
<tr>
<td>26</td>
<td>2</td>
<td>10</td>
<td>Week 1+3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>30</td>
<td>13</td>
<td>0.48</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>34</td>
<td>28</td>
<td>0.40</td>
</tr>
<tr>
<td>27</td>
<td>3</td>
<td>10</td>
<td>Week 1+3 Outside</td>
<td>Week 4 Outside</td>
<td>62</td>
<td>29</td>
<td>10</td>
<td>0.47</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Week 5 Outside</td>
<td>86</td>
<td>34</td>
<td>30</td>
<td>0.40</td>
</tr>
</tbody>
</table>
Attacks Detected and Detection Rate

From the results presented it can be observed that the amount of real positives detected vary with the difference in node limit. The largest amount of real positives detected is found the trees that are generated with a node limit of 5% - and particularly the ones trained with only week 1.

The amount of false positives changed in accordance with the set node limit. A node limit of 5% provide for a moderate amount of false positives. The trees generated with a node limit of 1% have a lower detection rate, and also showed an increase in the amount of false positives generated. Finally, the trees generated with a 10% node limit generate less false positives than in the other cases, but are also the ones with the poorest detection rate.

Given that only the outside data from the DARPA set was used, the attacks that occur only on the inside were undetectable by Inspectobot. Also undetectable are the locally executed attacks that do not generate any network traffic. When subtracting these attacks, only two of the undetected attacks are in fact undetectable in the outside sets: guesspop and one instance of land.

Based on the amount of detected attacks a detection rate has been derived from the results. The detection rate is calculated from real positives divided by the total number of attacks possible to detect.

From the results it can be observed that the best detection rate is found in the second tree generated of week 1 with a 5% node limit. During testing with week 4 80% of the attacks were detected and in week 5 70% were detected.

The results show that during testing with week 4 the detection rate varies from 50% to 82% based on the node limit used in the tree generation. It can also be observed that the node limit remains close within the different node limits, showing a consistency in the system. In week 5 the results varies from 36% to 70%, showing a slightly lower detection rate, but the consistency remains close.

Summary

In this test case, the 1999 DARPA IDS Evaluation Sets are used to determine if Inspectobot is able the listed attacks and how many false positive that are generated. In addition, the detection rate of the system is determined to provide an overview of the systems accuracy.

From the results acquired the system proves reliant and consistent with a top detection rate of 80%.

May 15th, 2011
Vegard Haugland
Marius Kjølleberg
Svein-Erik Larsen
Appendix C - GANTT Chart

During the course of the project, the following GANTT chart has been used to illustrate the project schedule. The work on the project started in January, and finished by the end of May.
<table>
<thead>
<tr>
<th>HierarchIndex</th>
<th>Name</th>
<th>Start</th>
<th>End</th>
<th>2010</th>
<th>2011</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Jan</td>
<td>Feb</td>
</tr>
<tr>
<td>1</td>
<td>Preliminary Project Complete</td>
<td>Mon 2011-01-03</td>
<td>Mon 2011-01-03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Report</td>
<td>Mon 2011-01-03</td>
<td>Tue 2011-05-24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>Chapter 1 - Introduction</td>
<td>Mon 2011-01-03</td>
<td>Sun 2011-01-30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>Preface</td>
<td>Mon 2011-01-10</td>
<td>Sun 2011-01-16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>Chapter 2 - Machine Learning and Applications</td>
<td>Wed 2011-01-19</td>
<td>Sun 2011-02-27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>Chapter 3 - Solution</td>
<td>Tue 2011-03-01</td>
<td>Fri 2011-05-06</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>Chapter 4 - Testing and Validation</td>
<td>Fri 2011-04-01</td>
<td>Mon 2011-05-09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.6</td>
<td>Chapter 5 - Discussion</td>
<td>Mon 2011-04-11</td>
<td>Thu 2011-05-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>Chapter 6 - Conclusion</td>
<td>Mon 2011-04-25</td>
<td>Thu 2011-05-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.8</td>
<td>Revision</td>
<td>Tue 2011-04-26</td>
<td>Mon 2011-05-23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.9</td>
<td>Abstract</td>
<td>Wed 2011-05-04</td>
<td>Thu 2011-05-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.10</td>
<td>First Full Draft</td>
<td>Fri 2011-05-13</td>
<td>Fri 2011-05-13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>System Development</td>
<td>Sat 2011-01-15</td>
<td>Tue 2011-05-03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>WP1 - Classifier - Basic</td>
<td>Sat 2011-01-15</td>
<td>Mon 2011-01-24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td>WP2 - Classifier - Hierarchical</td>
<td>Tue 2011-01-25</td>
<td>Wed 2011-02-09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>WP4 - Graphical User Interface</td>
<td>Wed 2011-02-02</td>
<td>Tue 2011-04-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>WP3 - Anomaly Detector</td>
<td>Fri 2011-02-11</td>
<td>Thu 2011-04-21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>WP5 - Graphing</td>
<td>Mon 2011-02-28</td>
<td>Tue 2011-03-15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>System Development Complete</td>
<td>Tue 2011-05-03</td>
<td>Tue 2011-05-03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>System Testing</td>
<td>Thu 2011-03-10</td>
<td>Mon 2011-05-09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>TC1 - GRIDAC Parameter Tuning</td>
<td>Thu 2011-03-10</td>
<td>Thu 2011-03-24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.2</td>
<td>TC2 - Classification Evaluation with Artificial Data</td>
<td>Sun 2011-03-20</td>
<td>Mon 2011-04-04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.3</td>
<td>TC3 - IDS Evaluation with Artificial Data</td>
<td>Fri 2011-03-25</td>
<td>Tue 2011-03-29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.4</td>
<td>TC5 - IDS Evaluation using Network Packets</td>
<td>Mon 2011-03-28</td>
<td>Mon 2011-05-09</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>TC4 - Classification Evaluation using Network Packets</td>
<td>Thu 2011-03-31</td>
<td>Tue 2011-04-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>System Testing Complete</td>
<td>Tue 2011-05-10</td>
<td>Tue 2011-05-10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>