uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

u Ottawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Reuben Smith

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Master of Computer Science

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Correlating Intrusion Alerts with Unsupervised Learning

TITRE DE LA THÈSE / TITLE OF THESIS

Nathalie Japkowicz

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Michel Barbeau

Stan Matwin

Gary W. Slater

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES /
DEAN OF THE FACULTY OF GRADUATE AND POSTDOCORAL STUDIES

CORRELATING INTRUSION ALERTS WITH UNSUPERVISED LEARNING

Reuben Smith

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the MCS degree in Computer Science

Ottawa-Carleton Institute for Computer Science
Faculty of Engineering
University of Ottawa

Your file Votre référence
ISBN: 0-494-14952-3
Our file Notre référence
ISBN: 0-494-14952-3

Abstract

Alert correlation systems attempt to discover the relationships between intrusion detection system (IDS) alerts to determine the motivation of attackers. IDSs are deployed to detect computer attacks against a network, but the output of IDSs is considered low level since a single attack can be represented by several alerts. An alert correlation system enables the intrusion analyst to find important alerts and filter false positives more efficiently.

We present an alert correlation system based on unsupervised machine learning algorithms that is accurate and low maintenance. The system is implemented in two stages of correlation. At the first stage of correlation alerts are grouped together such that each group forms one step of an attack. At the second stage the groups created at the first stage are combined such that each combination of groups contains the alerts of precisely one full attack.

The first stage of our system uses an autoassociative neural network, the autoassociator, to discover correlations between intrusion alerts in the greater collection of alert data. We evaluated the autoassociator as a clustering algorithm, along with other clustering algorithms, in a series of experiments. We select the EM algorithm, a clustering algorithm based on separating statistical distributions, as the method for finding full attacks at the second stage.

We demonstrate our system using alerts generated by the Snort IDS. We discuss the overall performance of our system in comparison with a simple correlation algorithm. We attain an accuracy of 88.2% correlating alerts Snort has generated from the 1999 DARPA intrusion detection dataset, which we compare to the accuracy of 79.4% for the simple algorithm.

The system we present is the first system of alert correlation created based on unsupervised learning algorithms. Machine learning is difficult to apply to the problem of alert correlation because the attacks faced by a network change from month-to-month. Unsupervised learning is ideal because it can be applied without requiring an expert to update the system periodically, unlike in systems based on supervised learning.

Acknowledgments

I appreciate the direction and financial support of my thesis supervisor Dr. Nathalie Japkowicz and our research liaison at DRDC Dr. Maxwell Dondo. Thank you to the members of my thesis committee for reviewing my thesis.

Thank you to my family, friends and their pets for their love and support while writing this thesis.

Parts of this research work was funded by Defence Research and Development Canada (DRDC) through contracts W7714-3-08710 and W7714-04-09128. This work was published in two DRDC contract reports CR 2005-030 [1] and CR 2005-155 [2].

Copyright Notice

I acknowledge that parts of this thesis are based on research funded by DRDC. I acknowledge that Reuben Smith has included works copyrighted by DRDC. However, DRDC also states the following:

> The Scientific and technical validity of this Contract Report[1] is entirely the responsibility of the contractor[2] and the contents do not necessarily have the approval or endorsement of Defence R&D Canada.

Maxwell Dondo, PhD, PEng,
Scientific Authority (contracts W7714-3-08710 and W7714-04-09128)

[1] CR 2005-030 [1] and CR 2005-155 [2] in this case.
[2] Nathalie Japkowicz and subcontractor Reuben Smith in this case.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Intrusion detection systems (IDSs) are computer programs or hardware that attempt to detect attacks against a computer network.[1] IDSs are deployed to inform administrators of the threats against their network services and data. Attacks against networks are common and firewalls are suitable for stopping only certain types of attacks, so IDSs are important in protecting networks. IDSs report attack attempts as *alerts* that contain information about the attack to an intrusion analyst.

Correlating intrusion alerts means finding the context for a particular alert. This can mean finding alerts that are very similar to other alerts, it can mean finding that an alert could only be a consequence of an event represented by a different alert, and it can mean relating external information such as vulnerability information or network architecture to an alert.

All of these types of correlation are useful. By finding relationships of similarity or causality between alerts an analyst can deal with groups of similar alerts more quickly than if dealing with the alerts separately. Finding external information that relates to an alert can improve the chances of the correct detection of an attack. For instance, if a router reports that it received a malformed packet that caused a system fault this information could be correlated to a vulnerability report for a network router, which

[1]There exist IDSs that detect attacks against hosts as well, as we discuss in Section 2.2.

might give an analyst the clue that the router has been compromised.

In our research we are concerned primarily with the first type of correlation presented above: specifically, the task of finding correlations between IDS alerts that represent the same attack. We want to find relationships between alerts that indicate the motives and methods of a particular attack attempt against the network. IDSs can be configured to report all of the pieces of an attack, but they are often not capable of distinguishing which alerts are related to the attack or separating them from the massive amounts of unrelated alerts reported by the systems.

In our research we try to separate the alerts reported by an IDS into discrete clusters of alerts so that each cluster represents all the alerts for a particular attack attempt. For false alarms generated by the IDS, we want to cluster together related false alarms so that either the root cause of the false alarms can be understood and dealt with, or so that they can be analyzed or ignored collectively.

Table 1.1: Two correlated shellcode alerts

[**] [1:648:6] SHELLCODE x86 NOOP [**]	[**] [1:648:6] SHELLCODE x86 NOOP [**]
11/14-21:55:36.*566507* 129.118.2.10:57425->170.129.50.120:63414	11/14-21:55:36.*576507* 129.118.2.10:57425->170.129.50.120:63414
TCP TTL:51 TOS:0xA0 ID:*46490* IpLen:20 DgmLen:1420 DF	TCP TTL:51 TOS:0xA0 ID:*46491* IpLen:20 DgmLen:1420 DF
A* Seq:*0xA074E240* Ack:0x90CF9E29 Win:0x16D0 TcpLen:20	***A**** Seq:*0xA074E7A4* Ack:0x90CF9E29 Win:0x16D0 TcpLen:20

As a concrete, very simple example of how two alerts can be related, note Table 1.1. In this table you see two alerts from the same IDS. In fact these two alerts were generated only one one-hundredth of a second apart and they are identical in most characteristics. The alerts represent the possible detection of a buffer-overflow attack. Buffer-overflow attacks are a common type of attack that exploit programming errors in network services that allow remote execution by the attacker (see Section 2.1.1). They are sometimes detected by recognizing a segment of executable code in the payload of a packet. The two alerts in Table 1.1 differ in their capture times, IP ID and TCP sequence numbers.[2] While these alerts differ in some attributes, it is clear from their capture times, IP source and destination addresses and TCP source and destination ports that these two alerts are related. Our system of alert correlation discovers these types of correlations as well as more complex correlations to give more context to individual alerts.

[2]For information on these communication protocol attributes, see the background in Section 2.1.

1.1 Goals of the Research

In this thesis we are interested in discovering whether unsupervised machine learning algorithms, in particular data clustering algorithms, can be applied to the problem of alert correlation and whether it is practical to do so. Unsupervised learning algorithms discover relationships between similar data without the decisions of an expert to indicate how the data is similar. Unsupervised learning algorithms are employed in problems where the decisions of an expert are not available.

We are interested in applying unsupervised learning to the problem of alert correlation because using machine learning in alert correlation solutions can reduce the amount of maintenance needed for the system and because supervised learning is ill-suited to the problem. Alert correlation systems not based on machine learning must either require that an administrator updates rules of correlation as new attacks are found or form these rules automatically. Supervised learning is ill-suited to the problem of alert correlation because the composition of the set of alerts encountered by the system changes significantly from month-to-month [3]. Effectively, the hypothesis that supervised learning systems require — that past behaviour will predict future behaviour — is partially violated.

In this thesis we are also interested in exploring the use of an autoassociative neural network called the autoassociator as a clustering algorithm. The autoassociator has been successfully applied to one-class learning problems [4], and we are interested in whether the algorithm can be used to cluster unlabelled data. We are interested in whether this algorithm has any benefits over other clustering algorithms with which we experiment, in terms of performance or other algorithm traits.

1.2 Overview of the System

The system we present has two stages of correlation, as seen in Figure 1.1. After the first stage of correlation, individual steps of the attacks in the dataset will be clustered. This means that, for example, all similar reconnaissance probes for each of the attacks in the

dataset of alerts will be clustered. After the second stage of correlation, different steps of the attacks in the dataset will be clustered. So after the second stage of the correlation system each cluster will represent one attack in the dataset.

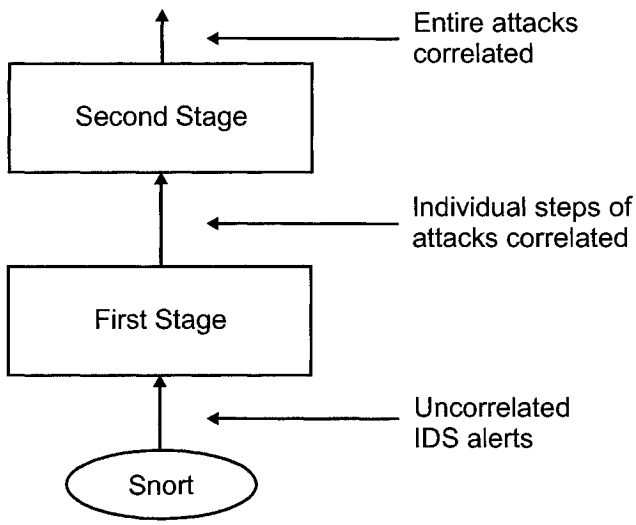

Figure 1.1: Architecture of our alert correlation system

At the first stage of correlation in our system we cluster the dataset of alerts by constructing features that differentiate Snort alerts at the IP packet level. We are interested in clustering alert IP packets that are numerically similar in their protocol attributes, so we construct our set of attributes to use for clustering in such a way that numerically similar Snort alerts are clustered together. At the second stage of correlation we are interested in the trends between groups of similar Snort alerts. For example, if two separate sets of Snort alerts are comprised of alerts with the same identical IP source address, this might be an indicator that the sets of alerts are related. The full details on how our alert correlation system is constructed are available in Chapter 3. The details of how alerts are encoded into a string of numbers so that they may be understood by the clustering algorithm at the first stage of correlation are in Section 3.2.1 and the details of how the resulting clusters are encoded at the second stage are in Section 3.3.1.

Neural networks are a type of machine learning algorithm useful in storing an abstracted form of large amounts of data in a novel way. Instead of storing all of the original data

unfiltered, neural networks are designed to store the trends of the data as weights within the network. The trends represented in a neural network are used to classify new data or to analyze data.

The autoassociator is a specialized neural network architecture which can be used to process data without labelled training data or supervision. The autoassociator's ability to learn from unstructured, unlabelled data makes it ideal to use for the intrusion detection alert data. Maintaining a supervised machine learning system can be costly because training data must be replaced and updated periodically, but the autoassociator does not require labelled data for training, so there is less need for maintenance.

Because the autoassociator can be trained on unlabelled data, the neural network can adapt to a particular stream of alerts easily. We give the details of how the autoassociator learns from alert data in Section 3.4.1 and we show how we interpret the autoassociator output to form discrete clusters in Section 3.4.2. The autoassociator is also highly customizable, which is important for the task at hand. We are interested in determining the suitability of other clustering algorithms as well, such as the EM algorithm and self-organizing maps (SOMs). The EM algorithm finds discrete clusters based on separating statistical distributions, and SOMs are single-layer neural network designed to visualize and cluster high-dimensional data.

We present the details and results of a number of experiments on our system in Section 4. We discuss the results of experiments such as determining the best clustering algorithms and combinations of parameters to choose for the two stages of our system and the effect of using different data scaling algorithms with the autoassociator. We also discuss the overall performance of our system, and how we calculated it, in Section 4.4. We conclude this thesis with Chapter 5.

1.3 List of Contributions

Alert correlation system based on unsupervised learning algorithms

In this thesis we have created the first intrusion alert correlation system based on

unsupervised machine learning algorithms. We have shown why an alert correlation system based on unsupervised learning algorithms can produce more realistic test results and can be more efficient in terms of operational maintenance. Specifically, we examined the suitability of different data clustering algorithms in our system and found that the autoassociator produced the best results at the first stage of correlation and the EM algorithm produced the best results at the second stage of correlation. The alert correlation system produced an accuracy 88.2% against a set of independent test data. We compared our system to a "naive" correlation algorithm based on descriptions in alert correlation literature that attained an accuracy of 79.4% on the same data. This naive correlation algorithm produced correlations based on time, syntactic and source proximity, similar to our system.

Using the autoassociator neural network as a clustering algorithm

We have examined the autoassociator neural network as an unsupervised machine learning algorithm to determine for which types of clustering it is best suited. We found that in the context of alert correlation in our system the autoassociator is well-suited to clustering alerts at the first stage of correlation, but is ill-suited to clustering sets of statistics about groups of alerts at the second stage of correlation. We considered different methods of interpretting the autoassociator output to form discrete clusters and we found that using a measure of Euclidean length of the error vectors produces the best results.

Chapter 2

Background

The IP and TCP protocols are the foundations of communication on the internet. These nonproprietary protocols allow semi-reliable digital communication between computers on networks connected to the internet. We discuss the TCP/IP protocols, particularly as they relate to intrusion detection, in Section 2.1. We discuss some common types of TCP/IP network-based computer attacks in Section 2.1.1.

As the internet has become more important to the general economy and in government systems, the risks posed by computer attacks against networks connected to the internet have increased. Intrusion detection systems (IDSs), as introduced in Chapter 1, are systems that detect computer attacks against a communication network. We provide an introduction to IDSs and discuss the current state of IDS research in Section 2.2. We discuss the most pertinent work in alert correlation in Section 2.3. We have a special report on machine learning-based alert correlation methods in Section 2.3.1 as well.

Machine learning is the problem of making computers learn [5]. In practice, this usually means making decisions about new data after examining old data. We discuss machine learning in Section 2.4. We give detailed background information on artificial neural networks in Section 2.4.2, and we talk about a specific, relevant type of neural network, the autoassociator, in Section 2.4.2 also. We give some information on two clustering algorithms with which we experiment, the EM algorithm and self-organizing maps, in

Section 2.4.3.

2.1 TCP/IP Communication Protocols

In TCP/IP Illustrated by W. Richard Stevens [6] we find all of the necessary information on *internet protocol (IP)* [7], *transmission control protocol (TCP)* [8] and other related internetworking protocols. We summarize the knowledge needed to understand the basics of the protocols that are commonly used on the internet. This is important because attacks on networks often exploit flaws in the protocols or require some in-depth knowledge of them to understand how the attack works. According to Northcutt [9], "[The Stevens book [6]] should be at the fingertips of any serious intrusion detection analyst." While our aim is not to train intrusion detection analysts, knowledge of internet protocols and attack methods benefit the reader of this thesis.

The internet protocols are a set of nonproprietary protocols that were designed by the United States government agency Defense Advanced Research Projects Agency (DARPA) in the mid-1970s. DARPA created the protocols because they needed a standard communication protocol for their packet-switched network of disparate systems. The protocols are well suited to both *wide area network (WAN)* and *local area network (LAN)* communications.

TCP and IP are considered lower-layer protocols in the OSI model for open system interconnection [6, 10]. Roughly, the TCP represents the transport layer and IP represents the network layer in the model.[1]

Figure 2.1 shows the layout of the protocol headers for a network packet communicated using the IP and TCP protocols. TCP is at a higher layer in the OSI model than IP, so, as indicated in the figure, TCP packets are encapsulated in IP packets. IP has two primary functions [7]:

[1]Technically other internet protocols, such as the *user datagram protocol (UDP)* [11], exist at these layers as well, and the *address resolution protocol (ARP)* [12] and *reverse address resolution protocol (RARP)* [13] exist to allow translation between the network and link layers, but the TCP and IP protocols are the most important internet protocols at these layers.

- IP provides for connectionless, best effort transmission of blocks of data between two endpoints (*IP addresses*) on the network, and
- IP provides for fragmentation and reassembly of blocks of data, to suit networks with different packet size capabilities.

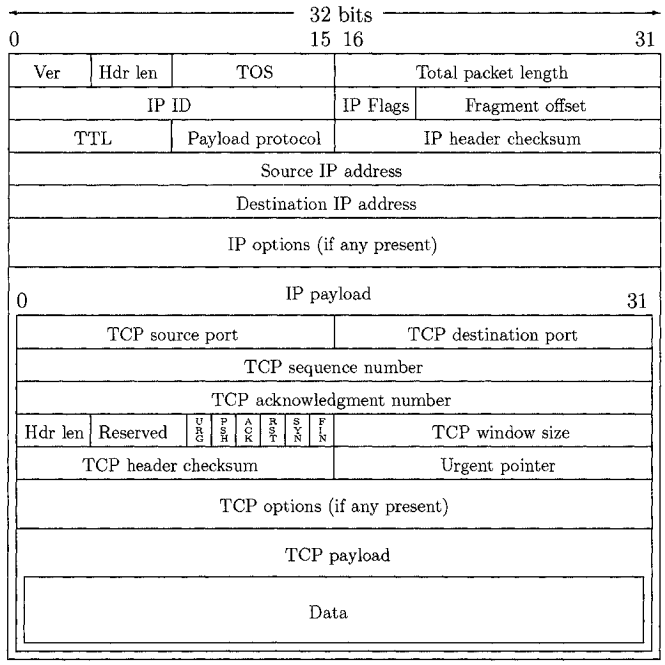

Figure 2.1: TCP/IP packet

The IP header fields shown Figure 2.1 are important to understanding how the IP protocol works. For each IP packet:

- *Version* — Describes the version of the IP protocol used.
- *Hdr len* (IP header length) — Indicates the number of 32-bit words in the IP packet header.
- *TOS* (Type of service) — Indicates the quality of service the packet is to receive, if this field is understood by the node forwarding the packet. The value for this field is typically specified by an upper-layer protocol and it can be used to distinguish traffic that requires a low latency, such as live internet telephony traffic.

- *Total packet length* — Specifies the length in bytes of the entire packet.
- *IP ID* (IP packet identification) — Designates an identifier for the packet. This field can be used to reassemble IP packets that were fragmented at a different network node by indicating which series of packet fragments the current packet is part of.
- *IP Flags* — Specifies two packet flags relevant to fragmentation. One flag indicates whether a packet can be fragmented and the other indicates whether this packet is the last in a series of fragments of the same larger packet.
- *Fragment offset* — Specifies the position of the current packet in a series of fragments so that the series can be reassembled to the original packet in the correct order.
- *TTL* (Time to live) — Gives a counter for the number of network node hops this packet can make before being discarded so that packets cannot be forwarded indefinitely if there is a misconfigured router on the packet's path. Each node decrements this value by one and the packet is discarded if this value reaches zero.
- *Payload protocol* — Indicates for which protocol the packet's payload is intended. In the case of the general packet in Figure 2.1, this value would correspond to TCP.
- *IP header checksum* — Gives a value that can help check the integrity of the IP header.
- *Source/Destination IP address* — Specify the network source node that sent the packet and the network endpoint node that is designated to receive the packet. IP addresses are visually represented in the form $a.b.c.d$ where each of a, b, c, and d are in the range $[0, 255]$. IP addresses are represented as 32-bit integers as $256^3a + 256^2b + 256c + d$ in *big-endian* byte order.[2]
- *IP options* — Specifies IP packet options such as security, if there are any.
- *IP payload* — Provides the data for the upper-layer protocol, in this case TCP.

[2]All integers in the IP packet are represented in big-endian byte order.

Intentionally left out of the IP standard are a number of important tasks, which are handled by TCP [8]: end-to-end reliability, flow control, and packet sequencing. Accomplishing these three tasks effectively makes TCP a reliable streaming protocol.

The TCP header fields of Figure 2.1 are important to describe the protocol. The *TCP acknowledgement number* and *TCP sequence number* fields are of particular importance in the protocol. It is through these fields that two network nodes can establish a streamed TCP connection and synchronize which data the other node has received. These numbers are initialized in the connection establishment phase and they are updated to reflect the amount of data passed between the network nodes over the duration of the connection. To establish a TCP connection, the two parties execute the *three-way handshake* [9], where **A** wants to establish a session with **B**:

1. **A** sends a TCP/IP packet with the TCP SYN flag set (a "SYN packet") and an initial sequence number specified to **B**.

2. **B** acknowledges **A**'s SYN packet by responding with a TCP/IP packet with the TCP SYN and ACK flags set (a "SYN/ACK packet") and the initial acknowledgement number specified to **A**.

3. **A** responds to **B**'s SYN/ACK packet and to finish the handshake by sending a packet with the TCP ACK flag set.

For an IP packet containing a TCP payload, the other TCP fields work as follows:

- *Source/Destination TCP port* — Specify the TCP port on the source network node and the TCP port on the destination network node of the IP packet. This is necessary so that many TCP connections can be establish concurrently between a pair of network nodes. The TCP port in a packet is sometimes useful in identifying which service is requested (port 80 sometimes indicates the *hypertext transport protocol (HTTP)*, for example), but it also can be chosen without regard to service by the operating system of a network node.

- *Hdr len* (TCP header length) — Indicates the number of 32-bit words in the TCP header.

- *Reserved* — Reserved for future use.
- *TCP Flags* — Controls the state of the connection. More than one flag may be set, but certain combinations of flags are illegal. See the table below.

Flag	Meaning
URG	The urgent flag indicates urgent data.
PSH	The push flag indicates that data is being sent in the packet.
ACK	The acknowledge flag indicates that data was received in a previous packet or that a three-way handshake is in progress.
RST	The reset flag indicates something is wrong and the connection is lost. This can also be used by an operating system to indicate that no service is listening at the port requested.
SYN	The synchronize flag indicates the intention to perform a three-way handshake.
FIN	The finish flag indicates that a party wants to end the TCP connection.

- *TCP window size* — Specifies the size of the receiver's incoming data buffer for flow control on the connection.
- *TCP header checksum* — Helps verify the integrity of the TCP header.
- *Urgent pointer* — The offset in bytes to the first byte of urgent data in the payload.
- *TCP options* — Specifies TCP options, if any are present.
- *TCP payload* — Provides the data that TCP is transporting. This is usually upper-layer protocol information such as part of an HTTP transaction.

2.1.1 Common Attacks on an IP Network

Several taxonomies of computer attacks exist [14, 15, 16]. These taxonomies seek to create categories for vulnerabilities that are logical and complete. We describe a few common or interesting computer attacks in this thesis. The categories of attack that we are interested in are the categories of attack that are common on IP networks. So

for our purposes, we refer to the categories of attack from the DARPA off-line intrusion detection evaluations [17, 18, 19], namely:

- *Probe, scan* or *reconnaissance* attacks — Lippmann *et al* describe this class as "attacks [that] automatically scan a network of computers or a DNS server to find valid IP addresses, active [TCP] ports, [the type of] host operating systems, and known vulnerabilities." Reconnaissance attacks are often the first step of an attack, followed by another type of attack if useful information is discovered by the attacker in the first step.

- *Remote-to-local (R2L)* attacks — Lippmann *et al* indicate "in these attacks, an attacker who does not have an account on a victim machine gains local access to the machine, exfiltrates files from the machine, or modifies data in transit to the machine."

- *User-to-root (U2R)* or *priviledge escalation* attacks — Lippmann *et al* describe this class as "attacks where a local [unpriviledged] user on a machine is able to obtain priviledges normally reserved for the UNIX super user or the Windows NT administrator."

- *Denial of service (DoS)* attacks — Attacks in this class try to shut down, degrade or disrupt in some way a host or network service remotely. A *distributed denial of service (DDoS)* attack tries to accomplish the same thing but by using multiple sources for the attack. DDoS attacks are considered a type of DoS attack.

Note that some attacks exist in multiple DARPA off-line IDS evaluation attack categories. Buffer overflow attacks, described below as a U2R attack, can be U2R, R2L, or U2R and R2L attacks, depending on the priviledges of the service compromised and on whether the attack can be executed remotely. A SYN flood, described below as a DoS attack, can be a reconnaissance attack masquerading as a DoS attack, or can be used as an R2L method to silence a host so that a trust relationship can be hijacked [9]. Clearly these the DARPA categories cannot be seen be canonical, but they provide a useful generalization nonetheless.

We look at examples of each of these four categories of attack to explain these attack categories in practice. We choose the attacks to be interesting as well as relevant to later parts of this thesis. We do not examine the defenses against these attacks here, although there are potent defenses for each of the attacks.

Reconnaissance Attack Examples

We look at two different reconnaissance attacks. In a *FIN scan* attack, an attacker sends a TCP/IP packet[3] with the TCP FIN flag set without having an existing TCP connection to elicit a response from a host or the network possibly behind a firewall, to determine whether an IP address is possibly assigned to a machine, or to determine whether a particular service is possibly available [20]. If the attacker receives a response that has the RST flag set, he can surmise that there is a computer at the destination IP address that he specified, but that the TCP port he specified is closed. If the attacker receives an *internet control message protocol (ICMP)*[4] message indicating "destination unreachable", then the attacker knows either that there is no host at the destination IP address or that a firewall between the attacker and the firewall is denying all traffic to the host. If the attacker does not receive any response to multiple identical FIN scans, the attacker knows either that there is a machine at the destination IP listening on the TCP port he specified or that there is a firewall between the attacker and the host that is filtering ICMP "destination unreachable" messages. This reconnaissance attack is considered stealthy because an IDS or firewall must keep a state of current connections with the network to know that the FIN scan packet is not part of a valid connection.

In a *FTP of .rhosts file* attack, an attacker, such as an insider to the network without advanced priviledges, tries to download the `.rhosts` file via a valid *file transfer protocol (FTP)* session [21]. The .rhosts file can detail trust relationships between various UNIX hosts on the network. The attacker can use this information later to exploit these trust relationships.

[3]Usually multiple FIN scan packets are sent to a range of hosts or services as part of the attack, similar to many scan attacks.

[4]ICMP is an internet protocol for control messages that uses IP to transmit data but exists at the network layer in the OSI model with IP.

Remote-to-Local Attack Example

An example of an R2L attack is an outside attacker attempting to guess the password of a known user on the network by repeatedly accessing the telnet or *secure shell (ssh)* services and guessing passwords automatically. By guessing passwords the attacker is trying to gain access to the known user account remotely.

This attack is called *remote password guessing* and it is an R2L attack because the remote attacker does not have any access to the host victim. This type of attack relies on network access from the attacker to the victim.

User-to-Root Attack Example

An example of a U2R attack is a *buffer overflow attack*[5] [22, 23]. The goal of the buffer overflow attack is to take control of a priviledged program, so that the attacker may access the same priviledges as the program [22]. The attacker takes control of the program by inputting a specially crafted string to the program that injects and executes attack code. The attacker is able to do this by inputting a string that is too large to fit into a buffer allocated by the program at a point in the program logic that does not correctly verify the length of the input string. The string overwrites the content of the buffer as well as some amount of memory after the buffer. If the buffer was allocated on the stack, the string can be crafted so that it precisely overwrites the return jump offset in the stack frame and, when the currently executing function returns, control is transferred to code supplied by the attacker in the same string [24]. The buffer does not have to reside on the stack, but that is the prototypical buffer overflow configuration which constitutes the majority of current buffer overflow vulnerabilities [22]. Buffer overflow attacks are most common in programs written in the C programming language because C does not perform implicit out of bounds checking on arrays. Buffer overflow attacks are commonly used by viruses to self-propagate on the internet.

[5]The alerts from correlation example in Table 1.1 are examples of a possible buffer overflow attack being detected by an IDS.

Denial of Service Attack Examples

We present two examples of DoS attacks. A *SYN flood* is a type of attack that can temporarily disable a remote host on the network with little effort if the remote host is not prepared for the attack [20, 25]. The attacker sends a number of initial connection requests to the victim (TCP/IP packets with the TCP SYN flag set, "SYN packets"), completing the first step of the three-way handshake of Section 2.1 for each connection. The victim acknowledges all of the connection requests it can, completing the second step of the three-way handshake for each incoming connection. But the victim must store the connection information, such as the remote IP address, the TCP ports, the new sequence and acknowledgement numbers, the initial window size requested, and possibly other information, for each connection. The victim has a finite number of slots for partially complete connections and it starts turning away connections and effectively ceases to communicate with the network after these slots are full. The victim eventually recovers from these half-completed connection attempts as the requests timeout, but the attacker can continue to send SYN packets to keep the victims slots full with relative ease. The attacker can also forge the IP header to make the attack seem like it is coming from a fake source, if he wishes to evade detection, since he does not need to receive any packets the victim sends.

In an *IDS flooding* attack, an attacker uses an attack tool to flood an IDS analyst's console with alerts, so that the analyst is overwhelmed, effectively performing a DoS attack on the operator of the IDS [26]. Two tools that perform IDS flooding attacks are *stick* and *IDSwakeup*. If the analyst is successfully overwhelmed by the attack, then the analyst might miss important, intentionally disguised attack traffic within the flood traffic.

2.2 Intrusion Detection Research

Intrusion detection systems were introduced in a seminal work by Denning [27]. There are many surveys of IDSs now [28, 29, 30] and Debar *et al* [28] provide a current taxonomy. There exist a number of books on IDSs [31, 32, 9], and we also draw from a brief, recent

introduction to IDSs by Kemmerer and Vigna [33].

The use of network intrusion detection systems is motivated by greater reliance on network services and connectivity, and risk of damage associated with said connectivity [17]. According to Lippmann *et al* [17]:

> It is difficult to prevent [all] attacks by security policies, firewalls, or other mechanisms. System and application software always contain unknown weaknesses or bugs[6], and complex often unforeseen interactions between software components and/or network protocols are continually exploited by attackers.

Network intrusion detection systems are systems which report packets or connections that are considered suspicious by some set of rules, or by other analysis against some pre-defined set of normal traffic. They exist to solve the problem of not knowing whether a target system or network has been compromised by attackers. They also function to alert administrators to the possible security vulnerabilities in their network. Cunningham *et al* [35] provide a good technical definition of an IDS:

> Intrusion detection systems monitor the use of computers and the network over which they communicate, searching for unauthorized use, anomalous behavior, and attempts to deny users, machines or portions of the network access to services.

Lippmann *et al* [17] add, "Intrusion detection systems are designed to detect attacks that inevitably occur despite [other] security precautions." We agree with the Lippmann *et al* perspective of IDSs. IDSs complement the use of firewalls in that they detect attacks that can penetrate firewalls. IDSs are useful in detecting certain types of attacks, but they also have failings, such as the lack of context reported with alerts and the high rates of false positives in out-of-the-box configurations of IDSs.

[6]Read Anderson [34] for an interesting analysis of these inevitable vulnerabilities.

Network versus Host Intrusion Detection

The term "intrusion detection system" can refer to either a *network intrusion detection system* or *host intrusion detection system.* A network IDS uses network traffic, usually IP packets, as its data source. It looks at the content and the header data of packets, searching for patterns or particular packet characteristics. A host IDS uses computer system log files or other system-based indicators, as its data source. Network IDSs detect a different set of attacks than host IDSs. Network IDSs are better at detecting attacks targetting the network itself, such as TCP port scanning or TCP hijacking, and seeing a broader view of attacks that take place [28]. Host IDSs are better at detecting priviledge escalation attacks and other attacks that do not show evidence on the network, such as attacks hidden with end-to-end encryption. The focus of this thesis is on network intrusion detection, so we use the terms "intrusion detection system" and "network intrusion detection system" interchangeably to refer to network intrusion detection systems.

Misuse versus Anomaly Intrusion Detection

There has been much written on the difference between *misuse* and *anomaly* detection systems [32, 28, 33, 36], an issue of how the IDS is implemented. Misuse detection, also known as *knowledge-based* or *signature-based* detection, uses knowledge of existing attacks to detect other similar attacks. Anomaly detection, also known as *behaviour-based* detection, uses the patterns of normal traffic to detect anomalous or possibly dangerous traffic.

Misuse detection systems use descriptions of known attacks to try to detect similar attacks[7], whereas anomaly detection systems use models of intended behaviour, whether trained or encoded by an expert, to interpret deviations from normal behaviour as attacks [33]. Each type of detection system has basic assumptions. A basic assumption in misuse detection systems is that attackers will not be able to vary the details of their

[7]Snort [37] is an example of a misuse intrusion detection system.

attack enough to subvert the IDS attack signature. A basic assumption of anomaly detection methods is that attacks differ from normal behaviour in some characteristic other than their goal.

One advantage that misuse detection systems have is that they produce results that are easily understood, or *interpretable*. Because each attack signature exists to detect an attack, it is possible to tell what attack is detected from what signature generated the alert. Another advantage of misuse detection systems is that they can potentially be configured to produce few false positives [28]. A drawback of misuse detection systems is that they cannot detect new or unseen attacks without adding new attack signatures to the system. This incurs a maintenance cost on the operator of the IDS.

The main advantage of anomaly detection systems is that they have the potential to detect attacks involving new vulnerabilities. The main disadvantage cited against anomaly IDSs is their high false positive rate. According to Debar *et al* [28], IDSs based on anomaly detection techniques have high false positive rates because "the entire scope of the behaviour of an information system may not be covered during the learning phase [of the anomaly-based system]."

Systems implementing anomaly-based intrusion detection research typically train on labelled network traffic that does not contain intrusions or abnormal traffic. The network traffic is labelled for the machine learning algorithm which processes the traffic. These systems are tested with traffic containing a small proportion of intrusions to normal traffic. There has been extensive research in recent years on this framework for anomaly detection [38, 39, 40]. In particular, Lee and Stolfo [39] compare the results produced from misuse detection systems to those of anomaly detection systems.

Deployment Location of IDS

The topology of the network where the IDS is deployed can be relevant to the performance of the system. IDSs can be deployed inside or outside the network's firewalls depending on the type and the level of detail of an attack that an administrator wishes to see. If an IDS is deployed outside the firewall then all attack attempts that do not penetrate

the firewall can be reported along with the attacks that penetrate. If an IDS is deployed inside the firewall then only attacks that are more likely to succeed are reported. IDSs are generally considered to be low-level tools in analyzing network traffic because of the vast number of possible attacks they report, and because they do not always accurately indicate to the analyst the context of the supposed attack, but if this information can be correlated then using information from both sides of the firewall can be fruitful.

2.2.1 False Positives

False positives are a significant research topic within the field of intrusion detection systems. More specifically, reducing false positives without impacting performance is a common topic. A *false positive* or *false alarm* is when an IDS reports that an attack is taking place when there is no such attack. False positives can be the product of a poorly configured signature-based IDS, when it has signatures that are too general to match specific attacks, or they can be the product of an anomaly detection system with the accepted false positive rate set too high. This said, false positives are accepted as an operating cost of IDSs. Northcutt [9] suggests, "as a general guideline for any network-based intrusion detection system [...] about 90 percent of all detects are false positives."

Table 2.1: Decision by IDS given event versus real status of event

	Actual attack event	Non-attack event
Generate intrusion alert	True positive	False positive
Ignore event	False negative	True negative

To understand false positives more completely it is important to understand what they exist in contrast with. Table 2.1 shows four possible outcomes when an IDS processes an event.

- If a real attack is reported by the IDS, this is called a true positive. Clearly this number must be more than trivially greater than zero (for a network is attacked) if the IDS to be considered valuable.

- If an IDS reports an event as an attack when the event is not actually an attack, this is called a false positive. If this number is too great then analysts will spend too much of their time analyzing events of little importance, so clearly this number is best if low.
- If a real attack is not reported by the IDS, this is called a false negative. When false negatives occur attacks against the network might remain unknown, which can be very damaging.
- If a non-attack event is not reported by the IDS, this is called a true negative. A true negative is a success because this is the correct behaviour for an IDS.

The problem with minimizing or maximizing one of these statistics without regard to the others is that the four statistics in Table 2.1 do not vary independently. An IDS could boast a *detection rate* (true positive rate) of 100% if it flags every network event as an attack. Alternatively, an IDS could have a false positive rate of 0% if it generated no alerts at all. But of course both of these extreme systems would be ineffective IDSs.

Receiver operating characteristic (ROC) curves are a tool used to analyze the interplay between two of the statistics from Table 2.1 visually [41, 42]. For example, true positives can be reported against false positives so that an IDS operator can choose an acceptable false positive rate given a detection rate, or *vice versa.*

A good example of an alert that has a reasonable probability of being a false positive is either of the alerts from the correlation example in Table 1.1. These "SHELLCODE x86 NOOP" alerts were generated by a Snort [37] signature that detects strings of hexadecimal `0x90` bytes. This sequence of bytes is common in undisguised buffer overflow attacks.[8] But this Snort signature can easily generate false positives as well, because strings of `0x90` bytes can exist in other network traffic as well, for instance if an uncompressed image is being transmitted. Signatures can be crafted in such a way to avoid false positives, but this signature is an example of a signature that will generate some false positives.

[8]This sequence of bytes is common in buffer overflow attacks because the hexadecimal byte `0x90` represents the Intel x86 "no-operation" opcode that exploiters of buffer overflow vulnerabilities use in constructing their attack strings. This sequence of opcodes is useful so that their addressing calculations do not have to be precise and so their attacks have a greater chance of succeeding.

2.3 Alert Correlation Research

Intrusion detection systems have a number of weaknesses, which motivates alert correlation research [26]. Debar [26] lists four weaknesses of IDSs in particular:

1. *Flooding* — IDSs are prone to generating a large number of alerts, whether true positives or false positives, which can overwhelm the IDS analyst. An attack on the IDS in Section 2.1.1 shows that an attacker can intentionally flood an IDS with alerts.

2. *Context* — IDSs generate many related alerts but do not necessarily show the relationships between these alerts. Logically grouping sets of related alerts helps the analyst by providing more information for each individual alert.

3. *False alarms* — As we noted in Section 2.2.1, often a significant proportion of the alerts generated by an IDS are false positives. False positives limit the effectiveness of an IDS because they require an analyst's time to handle them.

4. *Scalability* — Debar asserts, "Current intrusion detection architectures make it difficult to achieve large-scale deployment."

An alert correlation system refers to finding relationships between intrusion alerts and other sources of information, including other intrusion alerts, to support the interpretation and analysis of alerts by an IDS analyst. The "other sources of information" can vary dramatically. Some sources of information used by other alert correlation systems are:

- *Network environment* — A common suggestion of alert correlation systems is to include a network scanner with the system so that a correlation system can relate network information, such as which services are running on the network, what access the firewall allows, the routing architecture of the network, *et cetera*, to alerts [43, 3, 44, 45]. This type of correlation is useful because it can help assess the impact of an alert. So, for instance, the analyst can analyze an alert with priority if it involves a service known to be available on the network.

- *Known vulnerabilities* — Reporting sources such as CERT [46] and the *bugtraq*

mailing lists [47] list new publically-known vulnerabilities constantly. If vulnerability information can be matched to an intrusion attempt, the analyst can discover the plan of the attacker and the effects of the attack, such as whether a service was compromised [48, 49].

- *Alerts from the same alert stream* — One stream of alerts from an IDS can overwhelm the analyst. For instance, a single scan attack can generate thousands of alerts, since IDSs are "low-level sensors" [50]. To create a clearer view of the threats presented in a stream of alerts, it is important to group alerts that are part of the same attack [51, 52, 53, 36, 3].

- *Alerts from a different alert stream* — Larger security-conscious organizations sometimes deploy multiple IDSs at different places in the network topology to combat the problem of insider attacks, or to gain more perspective on attacks [54, 26, 52, 50]. Sometimes IDSs from multiple vendors are deployed as well, since different IDSs detect different attacks. Correlating these sources of alerts can produce a comprehensive view of the threats posed against a network.

- *Event reports from other sources* — Relating events, such as a fault detected at a firewall or router, to an intrusion alert can be a valuable tool in analyzing the validity of an alert.

Cuppens [36] states, "The objective of the correlation function is [...] to correlate alerts in order recognize the intrusion plan that is currently executed by the attacker." Haines *et al* [50] give the example that "a typical attack might involve a network *surveillance step*, followed by an *intrusion step* using a known vulnerability, followed by a *priviledge escalation step* to improve access to the target, and finally achieve some *goal step* such as information theft or denial of some system service." To compile these steps into a report about the complete attack for the IDS analyst requires the use of an alert correlation system.

There is also a field of research known as *intrusion event correlation* or *network event correlation*. This problem focuses on finding relationships between neutral events from all available sources [31, 55], whereas alert correlation focuses on finding relationships between intrusion alerts and other sources, including other intrusion alert sources.

Some correlation work focuses on the reduction of false alarms as a by-product of alert correlation [52, 44, 3, 50]. All effective alert correlation systems can aid in the faster analysis of false alarms, though. If a steady stream of false alarms are being generated by a particular source, then correlation will indicate to the analyst that a new alert is similar to the set of false alarms, so the new alert can be analyzed with lower priority. This approach can be argued to work for intentional flooding of IDSs with alerts too, because tools to flood an IDS automatically will leave clues in the traffic they generate.

Hätälä *et al* [56] reviews and criticizes the various alert correlation efforts of different research groups [52, 49, 53, 51, 36, 26]. This review is valuable because there have not yet been any surveys of the field of alert correlation published. They also discuss the *Intrusion Detection Message Exchange Format (IDMEF)* standard by the *Intrusion Detection Working Group (IDWG)* [54] which hopes to encourage the development and interoperability of alert correlation between deployed IDSs. The messages defined by the standard are intended to be passed between IDSs and between a correlation system and an IDS, as a standard method of communication between these systems. The standard specifies message classes such as Target, Source and [Alert] Classification. For each of these classes the standard specifies a number of necessary fields, such as which network node the class refers to, the service affected, and the name of the alert as given by the IDS. However, Hätälä *et al* offer this criticism of the standard:

> Unfortunately, no global scheme for event classification exists and therefore the Classification component of the Alert is not globally useful for correlation. The correlator has to have knowledge about specific attack strings that the messages carry and the participating IDS systems [sic] must all use the same vendor-specific or otherwise agreed upon scheme, a scenario that is not very likely in heterogeneous networks.

We agree with the Hätälä *et al* criticism of the IDMEF standard. The type of alert correlation envisioned in the creation of the format is too narrow. The standard ignores too much information reported by the IDSs that could be useful in alert correlation, even though alert correlation was one of the motivations for the standard.

2.3.1 Machine Learning-Based Alert Correlation Systems

Alert correlation systems are sometimes based on machine learning. (See Section 2.4 for an introduction to machine learning.) Systems of this sort rely on translating characteristics of an attack, reported as an IDS alert, into numerical tuples. Each component of the tuple created represents a characteristic of the alert that can be measured for every alert. Alerts, in their tuple form, are then classified, clustered, matched, or prioritized based on their similarity to other alerts. The characteristics chosen to represent an alert determine what relationships between alerts are discovered and how the relationships are organized.

Ning *et al* offer a criticism of numerical correlation techniques [45]. They criticize these systems as not being an effective correlation tools in general because they assert that often the relationships of alerts that a correlation system should find cannot be represented by numerical similarities. They also state certain approaches "cannot discover the causal relationships between related alerts that do not share similar features" [45, 56]. As we presented in the introduction, though, causal relationships are not the only type of alert correlation that is useful. We also note that non-numerical correlation techniques can suffer a different set of problems. If the logic of the non-numerical system tries to match numerical attributes of an attack, the system will not match attributes that are close but not exactly equal correctly.

One system that uses machine learning techniques to perform alert correlation on a stream of alerts is by Julisch and Dacier [3]. They describe a method for alert correlation that uses a conceptual clustering algorithm of their own creation. They choose to use a conceptual clustering algorithm because the clusters produced are intrepretable and because these clustering algorithms are "good at handling categorical attributes such as IP addresses, [TCP] port numbers, and alarm types" [3]. Julisch *et al* use a variant of *attribute-oriented induction (AOI)* [57] as their conceptual clustering algorithm. AOI uses hierarchial models of the clustering attributes to assist in clustering. The algorithm iteratively generalizes attributes until all attributes are generalized to some threshold. At each step of generalization, alerts that become identical after being generalized are merged. Julisch *et al* propose changes so that over-generalization of alert attributes does

not occur, which could lead to unrelated alerts being clustered. In their experiments they find that 75% of alerts can be analyzed automatically using their system.

Julisch *et al* resolve the problem of training their system by relying on clustering algorithms rather than classification algorithms. However, they require some training in their system to adapt it to the stream of alerts so that some alerts can be analyzed automatically, since their system explicitly has the goal to reduce the number of alerts presented to the administrator. Training their system fits into the general framework they have created for IDS operation. They advocate tasking an administrator with modifying the configuration of the correlation system by hand once per month to reflect the changes in the network environment. As well, in their system they require that the correlation system has information specific to the local network before initial operation of the system. They suggest that a seasoned analyst performs this initial configuration.

We contend that the maintenance burden of the Julisch *et al* system, combined with the expertise required in configuring the system, limits the practicality of their system. While some larger organizations will employ experts in intrusion detection and alert correlation, there are many organizations that rely more on automation and non-expert administrators to configure and watch their IDSs.

Dain and Cunningham present another alert correlation system to determine the attacks within one or more streams of alerts [51]. They group together all of the alerts relating to a single attack into *attack scenarios* using machine learning algorithms. In their work, Dain *et al* train machine learning algorithms such as neural networks and decision trees to recognize attacks based on a novel list of alert attributes. The Dain *et al* system uses a labelled training dataset to train a classifier on relationships between alerts so that these relationships can be encoded in the classifier. They use the three most recently reported alerts to predict in which group of alerts the current alert should be placed. Their hope is that patterns or sequences of alerts will be common and useful in predicting the placement a subsequent alert. Dain *et al* also use a number of other attack characteristics, such as source and target IP addresses, whether a machine is already believed to be compromised, and whether the most recent alert of a group of alerts is of the same type as the current alert. They train their classifier on a training dataset, they use an evaluation dataset to discover optimal algorithm parameters for the classifier, and they test their trained and

optimized classifier on a test dataset to generate their statistical results.

Hätälä *et al* [56] criticize the Dain *et al* work for its use of a simplistic dataset, and Dain *et al* [51] mention this fact as well. The Dain *et al* research is tested only with a DEFCON conference dataset [58]. The attackers represented in the dataset are motivated by points in the competition and points are not awarded for stealthy attacks [51, 56]. Also, a very high proportion of the traffic in the dataset is attack traffic because of the nature of the contest. Lastly, many of the attacks in the dataset originate from a single IP address [51]. The use of this dataset simplifies the problem of alert correlation because the attacks are more trivially correlated and because false positives are less likely to affect results.

We present here a criticism of the methodology of the Dain *et al* work. In the explanation of their preparation of the dataset, they offer the detail that, "a script was written to randomly split the data into [training, evaluation and testing] files." This statement implies that care was not taken to ensure that all training and evaluation instances occurred before any testing instances, and in fact that it is very likely that many testing instances were derived from alerts occurring before training instances. This is a methodological problem because it does not model a real world realtime system (which their system is purported to be), where the future parts of an attack are unknown. As well, because the authors split up individual attack scenarios over the training and testing datasets, the system was trained on one part of an attack scenario then tested on a different part of the same attack scenario, which is implausible for any realistic training set. (To mimic the paper's test configuration in the real world an analyst would have to label one third of the alerts in an attack before the system could classify the other two thirds.) Lastly, even if the above were untrue, Dain and Cunningham would still have the problem that they trained and tested on the same dataset, implying that in any situation where their system would be deployed, significant work would have to be undertaken to label a large set of training data by hand [45]. And since they used the same dataset for training, evaluation and testing, it is possible that their classifier is overfitted to the dataset [56]. Their excellent performance in the domain occurs because their tests were unfairly biased towards success.[9] In fact, because of these methodological errors, their testing results

[9]The Dain and Cunningham system attained an improbable performance level of over 99%. Another reason this figure is so high is the way they formulated their experiment. The machine learning algorithm gets credit for many correct decisions not to place alerts into existing scenarios, even when it has placed

can be discounted, regardless of the novel ideas they present in their paper.

The work in this thesis is presented in other papers and reports [1, 2, 59]. In particular, neural networks are examined for suitability as an alert correlation agent in these works.

2.3.2 Non-Machine Learning Alert Correlation Systems

There are a number of approaches to alert correlation that are not based on machine learning techniques. Because this set of approaches can involve statistical methods, logical rules, graph theory algorithms or other methods, this group of systems cannot be summarized easily. The methods in this section also provide different types of correlation information and possess different system goals. We summarize a number of important alert correlation systems here individually.

Ning and Cui [45, 53] use the idea that some alerts happen as a consequence of other alerts as the basis for their system, which they implement using correlation rules. Ning *et al* recognize that many existing IDSs tend to detect low-level attacks [45] without being able to relate these alerts to the intentions of the attacker. Their off-line system produces correlations from the stream of alerts of an IDS. The goal of their research is to find sequences of alerts that signify the plan of an attacker using prerequisites and consequences for each type of alert. Ning *et al* [45] give an example of these relationships: "a certain network born [sic] buffer overflow attack may require that the target host have a vulnerable service that is accessible to the attacker through the firewall." They find these sequences using a logic-based system that assigns prerequisites and consequences to each type of alert, allowing them to compute a *hyper-alert*, or amalgamation of related alerts. They represent prerequisites and consequences as logical predicates such as *VulnerableToBOF(IP,port) AND AccessibleViaFirewall(IP,port)* to represent the previous example prerequisite. Ning *et al* compute these hyper-alerts by *joining* a table of consequences with a table of prerequisites in a *SQL* query. Computing these hyper-alerts allows the analyst to see the consequences of all alerts and analyze the hyper-alerts with most devastating consequences first. They demonstrate the usefulness of their system by

the alert in question into an incorrect scenario.

showing how their system can significantly reduce the number of false positives reported by an IDS while negligibly reducing the number of true positives.

Valdes and Skinner [52] present a system for using statistical techniques in the alert correlation problem. They present a system to group together alerts from heterogeneous network sensors. They control how loosely their grouping algorithm works with a single system parameter. In their system they represent alerts as tuples of numbers, like in Section 2.3.1, where each number represents a characteristic of the alert. The characteristics chosen to represent alerts in this work are attributes such as source IP address and destination TCP port. Alerts are compared and checked for similarity using a distance measure where distances between commonly similar alerts are scaled shorter.

The approach of Valdes and Skinner is based on a layered approach to correlation. They first advocate reducing the number of alerts in a particular stream of alerts from a single sensor by grouping similar alerts. (This process is known as *threading*, though they do not name it as such in their paper.) Then they give details on how to fuse the results from different sensors into a single stream. And finally they show how to use their system to find correlations between the remaining clusters, where each correlation found represents a link between steps of an attack by an opponent. They create these three layers of correlation by adjusting statistical thresholds to find particular information at each layer. The criticism offered by Ning *et al* [45] in Section 2.3.1 applies to the Valdes and Skinner approach as well, because of their numerical techniques.

Haines *et al* [50] created a logic-based alert correlation system to find relationships between attacks from different IDSs and to extract the target information from a set of alerts. Their system combines the output of other correlation systems to create a system that performs better than any individual system. They used three dimensions of reasoning found in some of the correlators they tested: *prioritization* of alerts, *multistep correlation* to combine different alerts of the same stream, and *multisensor correlation* to combine alerts of different streams. If the correlators used in their testing match one of these dimensions within a certain threshold, the Haines *et al* correlation system considers the attacks relevant. Using this logic for both attack-recognition and target-identification purposes, they are able to correlate most relevant attacks and to reduce the number of alarms produced significantly.

Debar and Wespi [26] present an aggregation and correlation component (ACC) for an IDS. They use this correlation component to flag alerts as an original, a consequence, or a double alert. In this research, they present rule-based methods for attaining this goal. They concern themselves with implementation issues such as integrating their ACC with existing IDSs — namely the Tivoli Enterprise Console, and they suppose the use of the IDWG [54] format for their work — and they discuss other issues such as raising or lowering the priority of IDS alerts to reflect the inferences of their ACC.

Cuppens *et al* [36] present the CRIM system to "manage, cluster, merge and correlate alerts." In the CRIM system they cluster alerts that refer to the same attack by using features available in the IDMEF format [54] and merge the alerts into a single alert to represent "the information contained in the various alerts belonging to this cluster" [36]. They then correlate steps of attacks using prerequisites and post-conditions, similar to Ning *et al* [53, 45]. They model these attack types in an attack description language of their own creation, LAMBDA. Debar *et al* [60] report one theoretical problem with the CRIM system. Debar suggests, "attacker strategies are too random to be the subject of explicit attack scenarios," which means that any correlation system cannot use strict rules because attacks vary widely in how they are perpetrated.

The non-machine learning alert correlation systems in this section fail in their lack of flexibility in approaching the problem. The systems of Ning and Cui, Debar and Wespi, and Cuppens *et al* rely on updates as new attacks and new correlation paradigms emerge. The Haines *et al* system is useful in that it combines the output of multiple IDSs or even alert correlation systems, but the system also assumes certain goals of the IDS, such as target determination, and their rules for correlation are very simple and unnuanced.

The approach of Valdes and Skinner suffers from an over-reliance on certain features of IP packets, such as the IP source and destination addresses. Attackers launch different parts of an attack from different source IP addresses and sometimes seem to target many destination IP addresses to confuse analysts monitoring their activity. Also, Valdes and Skinner use probabilities of sequences of alerts to correlate that are deductively chosen, in contrast with the empirically determined sequences of Dain *et al*, which is an algorithmic weakness because these static probabilities are more likely to produce errors. The testing of the Valdes and Skinner is non-rigourous as well, which is another problem in their

research.

Lastly, the systems in this section differ somewhat in their goals, compared with the system of alert correlation we present in this thesis. The Haines *et al* system focuses on target analysis and the combining of the output of multiple IDSs. The Ning and Cui, Debar and Wespi, and Cuppens *et al* systems all try to report the consequence relationships between alerts. The Valdes and Skinner, Ning and Cui, and Cuppens *et al* systems focus more on the reduction of the alert load generated by the IDS.

2.3.3 Intrusion Detection Systems that Correlate Alerts

There are a number of deployed IDSs. Some are commercial systems and others, such as Snort, are available free of charge. We examine the common IDSs that offer alert correlation capabilities.

- EMERALD [43] is reported to be a distributed, scalable, hierarchial, customizable network intrusion monitor. It can be combined with other IDSs to correlate threat warnings between different systems. EMERALD explicitly divides statistical analysis (anomaly detection) from signature-based analysis (misuse detection), and combines the results in the correlation unit. The details of the correlation unit are given in Section 2.3.2 and Valdes *et al* [52, 56].

- Snort [37] is a lightweight, open source IDS. Snort flags network traffic that might be considered suspicious by matching traffic to a set of rules or signatures. Snort is able to store all incoming alerts in a SQL database and offers a packet-level view of the alerts it generates. Snort does not have a correlation unit but it is able to sort alerts by source IP address and ACID [61], a simple correlation tool, is designed to work with it. Snort has various modules to reconstruct TCP connections and the Snort installation is distributed with a default set of misuse detection rules. ACID is a web-based intrusion correlation console that offers only simple correlation abilities. With ACID, the IDS analyst can correlate alerts by source or destination TCP port, source or destination IP address or alert type by sorting based on these variables. This one- or two-dimensional analysis leads to problems when trying to

detect attacks such as distributed denial of service attacks or other multi-source attacks.

- Shadow [62] is an IDS created by Stephen Northcutt *et al.* In his book, Northcutt *et al* [9] addresses alert correlation, but only in a capacity similar to that found in ACID. He focuses on the manual producedures administrators might use to correlate alerts rather than how to correlate alerts automatically. He mentions correlation by alert arrival time, as well as by the other attributes listed for ACID.

- QuidSCOR [48] is an open-source IDS, though it requires a subscription from its publisher, Qualys Inc, to work correctly. It works with IDSs such as Snort to correlate alerts against Common Vulnerability and Exposure (CVE) information published by CERT [46].

From the list above, it is clear that alert correlation is not yet a widely deployed solution. The goals of alert correlation between deployed systems are very different, and the literature on these systems do not report performance of their systems or give sets of concrete examples of how their systems succeed or fail.

2.4 Machine Learning

Machine learning is a multidisciplinary field of research whose aim is to allow computers "to improve automatically with experience" [5]. Machine learning techniques are applied to specific problems to determine classifications of new data automatically. For example, for the problem of speech recognition, "algorithms based on machine learning outperform all other approaches that have been attempted to date" [5]. Machine learning systems are sometimes contrasted with expert systems. Expert systems make decisions based on sets of rules, usually created by experts, whereas systems based on machine learning examine and learn from the previous decisions of experts to make decisions.

Witten *et al* describe *machine learning* algorithms informally[10] as "techniques for finding and describing patterns in data" [63]. They also describe the process of machine learning

[10]For a formal treatment of machine learning consult Mitchell [5].

algorithms as searching through possible hypotheses to generalize training data. Machine learning algorithms typically function on multidimensional data. Each *data item* — also known as *instances* or *examples* — represents an "individual, independent example of a concept to be learned." Each data item is characterized by "its values on a fixed, predefined set of *features* or *attributes*." For example, in a medical dataset, each data item might represent a patient with cancer and each feature would be a characteristic of the patient, such as number of previous diagnoses of cancer, a measure of hereditary predisposition to cancer, body mass index, white blood cell count at diagnosis, whether patient smokes, and so on.

Machine learning, as a general field, offers different kinds of learning, but we are concerned with *supervised machine learning* and *unsupervised machine learning*. The training step in a supervised learning problem has access to a training set of data as well as a set of desired outputs, or class labels, to correspond to each training data item. The supervised learning algorithm takes advantage of the known target outputs to train the classifier to recognize similar data then classify it correctly. The training step of an unsupervised learning problem has access to a training set of data without a set of desired outputs. The unsupervised learning algorithm builds internal representations of the training data, which can be used to cluster similar data.

In practice the choice of which type of learning algorithm to use lies in what data is available and what the goals of the research are. Jain *et al* [64] contrast supervised learning algorithms with clustering algorithms, which are a type of unsupervised machine learning algorithm:

> In supervised classification, we are provided with a collection of *labelled* [data items]; the problem is to label a newly encountered, yet unlabelled, [data item]. Typically, the given labelled [data] are used to learn the descriptions of classes which in turn are used to label a new [data item]. In the case of clustering, the problem is to group a given collection of unlabelled [data] into meaningful clusters.

We discuss different datasets used in experimentation with machine learning algorithms in Section 2.4.1, and we introduce neural networks in Section 2.4.2 and clustering algorithms

in Section 2.4.3.

We include self-organizing maps in the section on clustering algorithms (Section 2.4.3) so we should note that self-organizing maps are neural networks and could have been discussed in Section 2.4.2. We employ them as a clustering algorithm here, so we describe them in that section with this note.

2.4.1 Training, Evaluation and Testing Datasets

We consider three types of datasets used in machine learning problems. Since most supervised machine learning algorithms classify based on previously seen classification examples, a training step is necessary for most algorithms. Classifiers learn how to classify new examples based on previously encountered examples. The *training dataset* is used to train how the classifier should classify new data. In an unsupervised machine learning problem, a training dataset with unlabelled examples can be used to train the clusterer to the types of data that will be encountered in new examples.

In both supervised and unsupervised machine learning problems the system can be *evaluated* or *validated*. We use the word *evaluate* to mean "to gauge the performance of, with the intent to attain better performance under the same conditions by modifying the classifier or clusterer in question." The *evaluation dataset* or *validation dataset* is a dataset used to evaluate a machine learning classifier or clusterer with an experimental set of parameters or components, with the knowledge that the evaluation is part of an effort to maximize the performance of the system for that dataset.

We contrast evaluation datasets with *testing datasets*, whose purpose it is to report on performance independently after optimization on the *evaluation dataset* has been completed. The data in the testing set should be as independent as possible from the training and evaluation datasets, within the parameters of the problem being solved. The testing set data should emulate data that might be encountered in a deployment situation, to allow for accurate measuring and reporting of performance. For instance, the testing set data should be selected from the most recent data items generated if generation time is relevant to the problem. Constructing the testing configuration in

this manner is important to model how the classifier or clusterer will be deployed. The classifier or clusterer will always be trained on existing data then applied to future data in practical applications.

2.4.2 Artificial Neural Networks[11]

The area of artificial neural networks (ANNs) have been a subject of intense research lately. As a result, the field of ANNs is now very broad and there are many books written on the subject [65, 66, 67]. In this section, we present a brief overview of ANNs.

ANN models take their inspiration from the human brain, through the interconnection of simple computational elements called neurons [67]. Each neuron is linked to some of its neighbours through connections of varying strengths. Learning is accomplished by continuously adjusting these connection strengths (weights) until the overall network outputs the desired results. These weight adjustments are based on mathematical algorithms used in solving nonlinear optimization functions.

The Neuron

The basic computational element of an ANN is called the neuron or processing node. The neuron model is based on highly simplified considerations of the biological neuron. A simple node is shown in Figure 2.2, where N inputs are summed at the node. Each

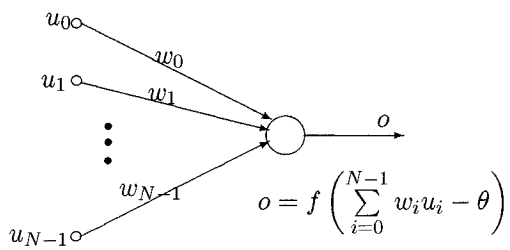

Figure 2.2: The basic neuron

input u_i is connected to the processing node through the synaptic connections, which are

[11]Note that this section is reproduced with permission from two reports for DRDC [1, 2].

represented by connection strengths called weights w_i. A bias term θ is also used at each node. The sum is fed through a transfer function f, called the activation function, to generate the output o. The signal flow is considered unidirectional as indicated by the arrows.

Although ANNs are constructed using this fundamental building block, there are significant differences in the architectures and driving fundamentals behind each ANN model.

The Activation Function

The activation function f plays a pivotal role in the functioning of the neuron. It determines the node output. As in Figure 2.2, the neuron output signal is given by:

$$o = f\left(\mathbf{w}^T\mathbf{u}\right) \tag{2.1}$$

where $\mathbf{w}$ is the weight vector defined as

$$\mathbf{w} \equiv \begin{bmatrix} w_1 & w_2 & \cdots & w_N \end{bmatrix}^T$$

and the input vector $\mathbf{u}$ is defined as

$$\mathbf{u} \equiv \begin{bmatrix} u_1 & u_2 & \cdots & u_N \end{bmatrix}^T$$

There are many different types of activation functions f to choose from, depending on the application [67, 68, 69]. Some of the commonly used activation functions are shown in Figure 2.3. These activation functions are the *hard-limiter*, the *threshold logic*, and the *sigmoid*. Since real applications are usually modelled as continuous functions, the most commonly used continuous activation function is the sigmoid.

Activation functions may be either unipolar, for positive output, or bipolar for output that may be positive or negative. For example, the bipolar sigmoidal activation function is defined as:

$$f(x) \equiv \frac{2}{1 + \exp^{-\lambda x}} - 1 \tag{2.2}$$

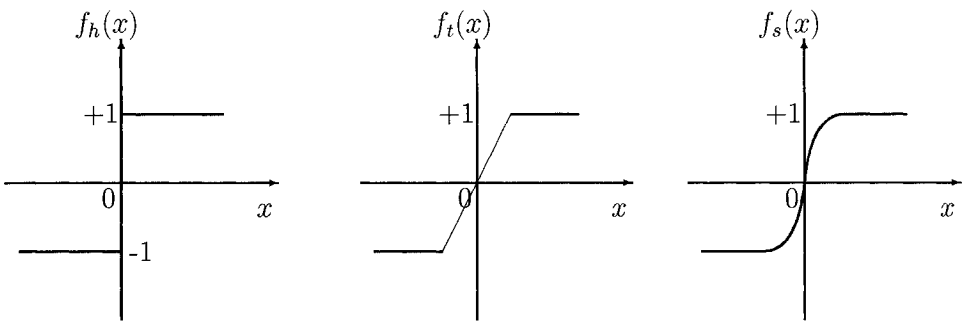

Figure 2.3: Activation functions

and the unipolar sigmoidal activation function is defined as

$$f(x) \equiv \frac{1}{1 + \exp^{-\lambda x}} \tag{2.3}$$

where λ is a constant.

A special case of an ANN is a single node based on the neuron model shown in Figure 2.2 and is called a *perceptron* after the work of Rosenblatt [67]. A perceptron consists of one or more neurons. If a continuous activation function is used, the neuron model is known as a *continuous perceptron.* A continuous perceptron is capable of classifying *linearly separable* classes of data of the form $a\mathbf{x} + b$. Multiple nodes in this format form a single layer multi-node ANN capable of classifying linearly separable data patterns.

Multi-Layer ANNs

ANNs are highly interconnected. ANNs are the simple clustering of primitive artificial neurons. This clustering occurs by creating layers of neurons which are connected to one another. Figure 2.4 shows a multi-layer perceptron. An input layer interfaces with the outside world to receive inputs and an output layer provides the outside world with the network's outputs. The rest of the neurons are hidden from view, and are called *hidden layers.*

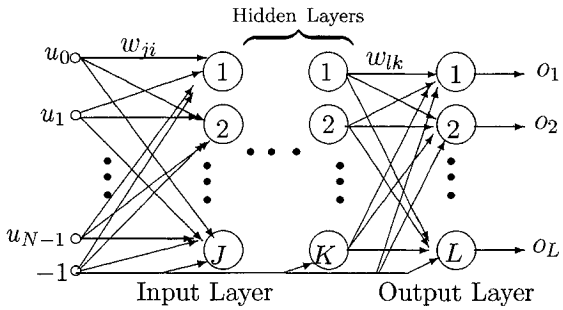

Figure 2.4: Multi-layer perceptron

The objective of using a multi-layer perceptron is to be able to classify patterns that linear classifiers (single layer ANNs) are incapable of classifying. The most important attribute of multi-layer ANNs is that they can learn to classify a problem of any complexity. The biggest challenge is usually in deciding the number of hidden layers in an ANN.

Zurada [69] gives an extensive discussion on the design of the number and size of hidden layers in a given architecture; nevertheless, trial and error methods have been widely used. If the number of hidden layers is too large, the ANN architecture will have problems generalizing; it will simply memorize the training set, making it useless for use with new datasets.

Inter-layer connections within an ANN architecture can take the following forms [69]:

- In a *fully-connected* ANN, each neuron on one layer is connected to every neuron on the next layer.
- In a *partially-connected* ANN, a neuron on one layer does not have to be connected to all neurons on the next layer.

If signal flow direction is taken into consideration, these two architectures can be further refined:

- In a *feedforward* ANN, the neurons on one layer send their output to the neurons in the next layer, but they do not receive any input back from the neurons in the next layer.
- In a *bi-directional* ANN, the neurons on one layer may send their output to the

next layer or the preceding layer, and the subsequent layers may also do the same.

- In a *hierarchical* ANN connection, the neurons of a lower layer may only communicate with neurons on the next level of layers.
- In a *resonance-connected* ANN, the layers have bi-directional connections, and they can continue sending messages across the connections a number of times until previously defined conditions are achieved.

The overall architecture of an ANN depends on the mappings required, the type of input patterns, and the learning rules to be used.

ANN Training

ANNs learn from experience by changing the ANN's connection weights until a solution is found. The learning ability of an ANN is determined by its architecture and by the algorithm chosen for training. The training methods [67] fall into broad categories:

- In *unsupervised training*, hidden neurons find an optimum operating point by themselves, without external influence.
- *Supervised training* requires that the network be given sample input and output patterns to learn. It is guided through the learning process until a satisfactory optimum operating point or a predefined threshold is reached. The most common training termination criteria is by setting a training threshold.

Backpropagation training is a form of supervised learning that has proven highly successful in training multi-layered ANNs. Information about errors is filtered back through the system and is used to adjust the connections between the layers, thus improving performance.

ANNs can be trained *on-line* or *off-line*. In off-line training algorithms, its weights do not change after the successful completion of the initial training. This is the most common training approach; especially in supervised training. In on-line or real time learning, weights continuously change when the system is in operation [69].

Training Rules

There is a wide variety of learning rules that are used with ANNs. Error minimization algorithms are used to determine the convergence levels when updating weights. In general, all ANN learning involves the iterative updating of the connection weights until the desired convergence is achieved. Most training algorithms start by initializing the weights to 0 or very small random numbers. This weight update is given by:

$$\mathbf{w}^{k+1} = \mathbf{w}^k - \Delta\mathbf{w}^k \tag{2.4}$$

Equation 2.4 is the ANN *general learning rule* [69]. The numerous learning rules, which are variations of this rule, only differ by the mathematical algorithms used to update the connection weights, or more specifically to calculate the value of $\Delta\mathbf{w}^k$ at each iteration k. Some of the common training rules are as follows:

- In the *Hebbian* rule [69, 68], the connection weight update $\Delta\mathbf{w}^k$ is proportional to the neuron's output. This was the first ANN learning rule [67, 70].
- The *perceptron* rule [67] updates the weights based on the difference between the desired output d and the actual neuron's response o.
- The *delta* learning rule [67, 70] is based on the minimization of the mean square error (MSE) as represented by the error function E, as shown in Equation 2.5.

  $$\mathbf{w}^{k+1} = \mathbf{w}^k - \eta\nabla E\left(\mathbf{w}^k\right) \tag{2.5}$$

  where η is a learning constant, and ∇E is the gradient of the error function E, defined by:

  $$E_k = \frac{1}{2}\left(d^k - o^k\right)^2 \tag{2.6}$$

  The objective is to iterate Equation 2.5 until the error E approaches zero (or a preset threshold value).

  For an ANN with P training patterns, and K outputs, the *root-mean square error*

(also known as the MSE [69]) is defined as:

$$E_{rms} = \frac{1}{PK}\sqrt{\sum_{p=1}^{P}\sum_{k=1}^{K}(d_{pk} - o_{pk})^2} \tag{2.7}$$

- The *Widrow-Hoff* [68, 69] learning rule (sometimes called the *Least Mean Square* learning rule) is considered a special case of the delta learning rule in that the neuron output o is independent of the activation function f.

- The most widely used supervised training approach which is derived from the Widrow-Hoff algorithm is the *error backpropagation training algorithm.* As the name implies, the error $\Delta\mathbf{w}^k$ is propagated back into the previous layers. This is done one layer at a time, until the first layer is reached.

  Consider an ANN with one hidden layer, K outputs, J hidden nodes, I inputs, and P training patterns. The output layer weights are adjusted as follows:

$$w_{kj} = w_{kj} + \eta\delta_{ok}y_j, \quad \text{for k} = 1, \cdots, \text{K}, \ \text{j} = 1, \cdots, \text{J} \tag{2.8}$$

  where η is a learning constant, y are the hidden units, and the output error $\delta_{\mathbf{ok}}$ is given by

$$\delta_{ok} = \frac{1}{2}(d_k - o_k)(1 - o_k^2), \quad \text{for k} = 1, 2, \cdots, \text{K} \tag{2.9}$$

  The weight update for the hidden layer is as follows:

$$w_{ji} = w_{ji} + \eta\delta_{yj}u_i, \quad \text{for k} = 1, \cdots, \text{K}, \ \text{i} = 1, \cdots, \text{I} \tag{2.10}$$

  where the output error $\delta_{\mathbf{yj}}$ is given by

$$\delta_{yj} = \frac{1}{2}(1 - y_j^2)\sum_{k=1}^{K}\delta_{ok}w_{kj}, \quad \text{for j} = 1, 2, \cdots, \text{J} \tag{2.11}$$

  The process is repeated for a number of iterations, or *epochs*, until a preset threshold of the MSE (Equation 2.7) is achieved, or for a set number of iterations until performance in an evaluation set is maximized.

Overfitting Avoidance

As the neural network is trained for more epochs, the neural network becomes more capable of recognizing the particular data on which it is being trained. One notable pitfall of the training process is that if the neural network is trained for too many epochs, then *overfitting* may occur. An overfitted neural network is a neural network that has been trained for more epochs than it should have for optimal performance. It describes the state of a neural network when it will no longer generalize to the testing or evaluation data, and thus cannot be used as a general classifier or clustering algorithm [71]. Overfitting describes any trained machine learning classifier or clusterer that "[depends] too strongly on the details of the particular examples used [in training]" [63].

To determine when optimal performance has been reached, it is necessary to measure the results of the classifier or clusterer against the evaluation dataset at periods of epochs. It is possible to recognize when the classifier or clusterer is optimally trained — that is, not overfitted or under-trained — by identifying the point where performance on the evaluation dataset begins to decrease.

The Autoassociator

The *autoassociator*, or an *autoassociative neural network*, is a feedforward, fully-connected, multi-layer neural network algorithm[12] [4]. The autoassociator recognizes one class of training data rather than discriminating between the trends of multiple classes of training data, as other neural network architectures are designed to function. The autoassociator is called an unsupervised learning algorithm because the classifier is trained on one class of data, so that it can recognize that class of data. Although the autoassociator is trained on only one class of data, it is tested against both classes of data in a *binary learning problem*, or learning problem with two classes.

In Figure 2.5, d is the number of *attributes*, also known as *features* or *components*, of the data in the dataset. The input values the attributes for a data item are represented

[12]Based on the work of Japkowicz [4], we extend the use of the autoassociator to a new application, a clustering algorithm. We use the autoassociator to cluster alerts outputted from an IDS. We give the details of workings this new clustering algorithm, and the motivation for it, in Section 3.4.1.

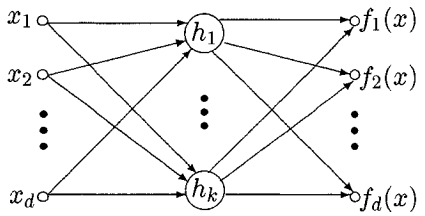

Figure 2.5: The autoassociator

by $x_1, \ldots, x_d$, the reconstructed output values of the autoassociator are represented by $f_1(x), \ldots, f_d(x)$, and the layer of k hidden units in Figure 2.5 is represented by $h_1, \ldots, h_k$.

The autoassociator uses the data inputs $x_1, \ldots, x_d$ as its training goals. Specifically, the neural network tries to make output values $f_1(x), \ldots, f_d(x)$ approximate input values $x_1, \ldots, x_d$. The training process effectively optimizes the weights in the ANN to be able to reconstruct test data that is common in the training data and effectively to fail to reconstruct the uncommon or other data. The potential of the autoassociator to internally represent data trends often lies in the number of *hidden units* and design of the hidden layer, so these parameters should be carefully tailored to the problem.

The autoassociator is a type of multi-layer perceptron (MLP), but it differs from the typical MLP of the **Multi-Layer ANNs** heading of Section 2.4.2. The most clear difference between the design of the autoassociator in Figure 2.5 and the typical MLP of Figure 2.4 is that the autoassociator has d output neurons, one for each input neuron, whereas the typical MLP has L output neurons, usually one for each output class in the labelled training data.

The *reconstruction error*[13] for a data item is computed using the $f_1(x)$, $f_2(x)$, ..., $f_d(x)$ autoassociator outputs shown in Figure 2.5. The formula for computing the reconstruction error E_x is:

$$E_x = \sum_{i=1}^{d} [x_i - f_i(x)]^2 \tag{2.12}$$

where x_i is the value of the i^{th} attribute of data item x and $f_i(x)$ is the value of the i^{th} output of the neural network with input x.

[13]This is sometimes called *squared reconstruction error* in other literature.

After training, the autoassociator is able to recognize the class represented in the training data because data items from the training class will have low reconstruction error values when compared to that of data items from unseen classes. The theory holds that data items from classes without representative training examples will have higher reconstruction errors.

The autoassociator is an off-line neural network algorithm that can be trained using the backpropagation algorithm.

2.4.3 Clustering Algorithms

Clustering algorithms are unsupervised machine learning algorithms that "apply when there is no class to be predicted but rather when the instances are to be divided into natural groups" [63]. The "natural groups" formed by the clustering algorithm are partially determined by the features available to represent the items to cluster. Clustering algorithms divide the data into natural groups by finding patterns in the data automatically.

Clustering algorithms can be trained on unlabelled data, or they can be applied to the test or evaluation data without training. Trained clustering algorithms build internal representations of the unlabelled training data during training, which they apply to the test data. Untrained clustering algorithms determine natural differences between subsets of data without prior insight into the data.

We present three clustering algorithms, the single-link clustering algorithm, the EM algorithm and self-organizing maps, in this section. In this thesis we also use the autoassociator, presented in Section 2.4.2, combined with the single-link clustering algorithm as a clustering algorithm, but this use is new, so we present it in Section 3.4.2.

Single-Link Clustering

A *single-link clustering algorithm*, or *nearest neighbour algorithm*, is a heuristic clustering algorithm that forms clusters based on whether a data item is within a minimum distance threshold of any alert in a cluster [71, 64]. We present this algorithm as Algorithm 1.

Algorithm 1 SINGLELINK(S, b)

Input: A set of data, S, on which the distance between two items can be computed.
Input: A threshold value, b, indicated the maximum distance between the two closest clusters in the output.
Output: A set, C, of clusters (sets) of the data in S.

1: $C \leftarrow \{\{c\} : c \in S\}$
2: // Note that $dist_{min}(X, Y)$ is the minimum distance between any pair (x, y), $x \in X$ and $y \in Y$.
3: **while** there exists a pair of clusters $(X, Y) \in C$ such that $dist_{min}(X, Y) \leq b$ **do**
4: $\quad (X, Y) \leftarrow argmin(\{dist_{min}(X, Y) : X, Y \in C, dist_{min}(X, Y) \leq b\})$
5: $\quad C \leftarrow C \setminus \{X, Y\}$
6: $\quad C \leftarrow C \cup \{X \cup Y\}$
7: **return** C

As presented in Algorithm 1, the clustering algorithm is known as a single-link algorithm because if the distance between any two data in a set is less than the threshold then those two data will appear in the same cluster.

The single-link clustering algorithm is often contrasted with the *complete-link clustering algorithm*, which requires that the distance between every pair of data in a cluster is less than the threshold. The $dist_{min}(X, Y)$ function in Algorithm 1can be replaced with $dist_{max}(X, Y)$ to make Algorithm 1a complete-link clustering algorithm, where $dist_{max}(X, Y)$ is the maximum distance between any pair (x, y), $x \in X$, $y \in Y$.

The EM Algorithm

The EM algorithm is a well-known clustering algorithm that was initially presented in Dempster *et al* [72]. The algorithm consists of two repeated steps, expectation and maximization. According to Witten *et al* [63], the EM algorithm uses a statistical model called *finite mixtures* to achieve the goal of producing the most likely set of clusters given the number of clusters, k, and a set of data. The model consists of a set of k probability distributions, one to represent the data of each cluster. There are parameters p_A, μ_A, σ_A^2 that define the distribution of cluster A, for each of the k distributions. The EM algorithm begins by making initial guesses for these parameters based on the input data,

then determines the probability that a particular data instance belongs to a particular cluster for all data using these parameter guesses. The distribution parameters are revised again and this process is repeated until the resulting clusters have some level of overall cluster "goodness" or until a maximum number of algorithm iterations is reached. The expectation step of the algorithm estimates the clusters of each data instance given the parameters of the finite mixture; the maximization step of the algorithm tries to maximize the likelihood of the distributions that make up the finite mixture, given the data.

Equation 2.13, Equation 2.14, and Equation 2.15 introduce the formulas used for estimation of the parameters for the distribution of cluster A. There are n items, $x_1, x_2, \cdots, x_n$, in the dataset to be clustered.

$$\mu_A = \frac{P[A|x_1]x_1 + P[A|x_2]x_2 + \ldots + P[A|x_n]x_n}{P[A|x_1] + P[A|x_2] + \ldots + P[A|x_n]} \tag{2.13}$$

$$\sigma_A^2 = \frac{P[A|x_1](x_1 - \mu)^2 + P[A|x_2](x_2 - \mu)^2 + \ldots + P[A|x_n](x_n - \mu)^2}{P[A|x_1] + P[A|x_2] + \ldots + P[A|x_n]} \tag{2.14}$$

$$p_A = \frac{|\{x : x \in A\}|}{n} \tag{2.15}$$

The calculation of μ_A uses the probability $P[A|x]$, the probability that data item x belongs in cluster A, for all x. The formulas for distribution parameters μ_A and σ_A^2 use the value $P[A|x]$ as a weight, so that data item x influences the computation of the parameters of A proportional to how likely x is in A. The formula for $P[A|x]$ is presented as Equation 2.16. The computation of $P[A|x]$ uses the normal distribution A probability density function $f(x; \mu_A, \sigma_A)$, in Equation 2.17, to estimate $P[x|A]$. Witten *et al* remark on Equation 2.16, "The denominator $P[x]$ will disappear: we calculate the numerators [$P[A|x]$ for all k distributions] and normalize them by dividing their sum.

$$P[A|x] = \frac{P[x|A] \cdot P[A]}{P[x]} \approx \frac{f(x; \mu_A, \sigma_A)p_A}{P[x]} \tag{2.16}$$

$$f(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.17}$$

The cluster "goodness" measure helps determine when the algorithm will finish. The calculation in Equation 2.18 estimates the overall likelihood that the data $x_1, x_2, \cdots, x_n$ came from this dataset, given the values for the k distribution parameters. Equation 2.18 is computed as a product over all data items x_i and features f.

$$\prod_{i,f} \sum_{A} p_A P[x_i|A] \tag{2.18}$$

The cluster "goodness" measure increases at each iteration of the EM algorithm [63]. A customizable heuristic is used to determine when the algorithm should end: If this "goodness" value increases less than the log-likelihood of 10^{-10} for 10 consecutive iterations, the resulting clusters are considered stable, so the algorithm terminates.

Self-Organizing Maps

Self-organizing maps (SOMs) are an unsupervised learning method useful in mapping a high-dimensional input to a two-dimensional output[14] such that similar inputs are mapped to locations close together in the map output [73, 74]. This mapping is primarily used for the visualization of high-dimensional data, but it can also be used for clustering. SOMs are a type of single-layer neural network.

According to Vesanto *et al* [75]:

> An SOM consists of neurons on a regular low-dimensional grid. Each neuron $[i]$ is a d-dimensional weight vector $[\mathbf{m}_i = [m_{i1}, \ldots, m_{id}]]$ where d is equal to the dimension of the input vectors. The neurons are connected to adjacent neurons by a neighbourhood relation [(such as a well-known distance measure)], which dictates the topology [...] of the map.

[14] Actually, other SOM dimensionalities for output are available as well, but two-dimensional output is the most common.

During training, the map adapts to the training data by moving the weight vectors $\mathbf{m}_i$ of the grid so that they span the topology and are organized. The training organization attempts to make similar training values appear in similar positions in the map. After training, the weight vectors $\mathbf{m}_i$ "define a tesselation of the input space into a set of Voronoi sets $V_i = \{\mathbf{x} : \|\mathbf{x} - \mathbf{m}_i\| < \|\mathbf{x} - \mathbf{m}_j\| \; \forall \; j \neq i\}$. In effect each [training data vector] belongs to the Voronoi set of the [weight vector] to which it is closest" [75].

It is recommended by Kohonen *et al* [74] that the training of the SOM is divided into two phases, an ordering phase and a tuning phase. During the ordering phase the training data are given their approximate correct position in the map, and during the tuning phase the positions of the data are fine-tuned. At each phase of training in MATLAB [68], the learning rate, training epochs, and neighbourhood function can be controlled.

Self-organizing maps are not natively intended for clustering, though they can easily be adapted to this application. There exist methods that to form clusters with SOMs [76, 77, 75, 68]. In the MATLAB implementation the SOM [68], the neural network is trained, then, as with other neural networks, data can be simulated on the network to discover the output of the network for that input. The neural network output of a data item input is a vector in the SOM. Any two vectors which share the same closest weight vector $\mathbf{m}_i$ in the SOM should have similar representation in the higher dimensional space, by the design of the SOM algorithm. For clustering the SOM output in MATLAB we can simply create one cluster for every weight vector $\mathbf{m}_i$ in the SOM output that has at least one mapped data item closest to it, then assign all of the testing data for the Voronoi V_i to that cluster.

Chapter 3

Our Model

Our model is based on unsupervised machine learning and is implemented in two stages of correlation. Our system finds relationships in a stream of alerts from an IDS. The purpose of our research is to develop a system of alert correlation that can find correlation between different steps of an attack. The first stage of the system, presented in Section 3.2, clusters alerts such that each cluster represents a different step of an attack. The second stage, presented in Section 3.3, clusters together different steps of a single attack. (Refer to Figure 1.1 from Chapter 1 for a visual representation of the system.)

Since our system is based on machine learning, a necessary step in the correlation process is to encode the alerts as numerical tuples. The first stage takes a set of alerts, encodes the first stage features of the alerts, and outputs clusters of those alerts. The second stage in our system takes the first stage output clusters, constructs the second stage features from these clusters of alerts, then outputs the correlations found by our system. The features used in the first stage are chosen such that each output cluster represents a distinct step of an attack. The features used in the second stage are constructed to find relationships between steps of an attack, so that the multiple steps of an attack are clustered together in the final output of the system.

If the alerts to be correlated by our system are false positives generated by the IDS, our system determines the relationships between these alerts as well. Each correlated group

of false positives should represent all similar false positives generated by one source. Our system is not designed to determine which correlated alerts are false positives, but an analyst can either ignore correlated false positives in groups — undoubtedly saving time — or an administrator can use the information to more accurately pinpoint the source of the false positives if it is a configuration or hardware problem.

Other approaches to alert correlation are presented in Section 2.3. We considered the approaches of Dain *et al* [51] and Julisch *et al* [3] in particular in the creation of our system. While these systems present some novel ideas on the creation of an alert correlation system implemented with machine learning, they have different goals and offer different services than the system proposed here. Both of these systems require some amount of training with labelled alerts or administrator supervision to correlate alerts. (See Section 2.3.1 for more details.)

Because the system we have created is based on unsupervised machine learning algorithms, less setup and maintenance are required in an operational setting when compared with a supervised system. The system we propose adapts itself to new network environments automatically by training on unlabelled alerts previously collected on the network. The composition of alerts reported on a network can change dramatically from month to month owing to different threats and attack trends on the internet [3]. Because of this state of flux, any system that requires significant operator input is impractical.

We discuss the datasets used for our examples, experimentation, and testing in Section 3.1. We present the details of our model in Section 3.2 and Section 3.3. We discuss the clustering algorithms used to implement our model in Section 3.4.

3.1 The Data

We use the Snort IDS [37] to generate alerts based on the stored traffic data, which was retrieved from the incidents.org website [78] and the 1999 DARPA IDS challenge dataset [18]. The alerts generated by Snort are output as text to Perl [79] scripts that parse the alerts and format the data for use with any machine learning algorithm.

3.1.1 Gathering and Selecting

We chose to use the incidents.org dataset for our evaluation because the dataset has a variety of interesting attacks reported by Snort and because the dataset was collected on a real network on the internet. There are many interesting correlations to be found in the incidents.org dataset, which suited our research because we wanted to demonstrate our system on real data. The dataset consists of real traffic captured on the internet — that is, the dataset was not generated automatically, but captured "in the wild" — which we believe accounts for the fact that it contains more interesting attacks to correlate. The 1999 DARPA dataset was generated automatically to model real traffic. Because of privacy issues it is only a simulation of real traffic. It is a standard dataset to use for testing IDSs and alert correlation systems, so we present our complete system tests and some preliminary evaluation results with this dataset. The DEFCON dataset [58] used in the Dain *et al* research was captured from a hacker competition where there was very little legitimate traffic on the network and where the attackers had no reason to hide their attacks. Because of this, and the fact that the dataset is no longer readily available on the internet, we chose not to use the DEFCON dataset.

The data we have selected offers the TCP/IP traffic as seen by only one IDS sensor. If we had access to traffic as seen by multiple sensors then we could have tested our system against this type of correlation as well, since we believe that similar alerts represented by different sensors will be encoded in the same way, but this type of data is not readily available to us, so we were not able to test our system with a multi-sensor configuration.

We select 10 000 alerts from each of the incidents.org and 1999 DARPA IDS datasets for training our system. When our system correlates alerts from the incidents.org dataset, the incidents.org set of training data is used to train the system. Similarly, we train using the DARPA dataset for correlation results on DARPA alerts. The alerts in these two sets of 10 000 data are unlabelled, which means that their true classification or clustering, as determined by an analyst, is not known and thus the data are not labelled with a class or cluster. We also select alerts for evaluation of our system from these two data sources, and we select alerts strictly for testing from the 1999 DARPA IDS dataset. We discuss the details of how these training, evaluation, and testing sets were chosen in Section 4.1.

3.1.2 Composition of DARPA Datasets

It is important to analyze the datasets we work with so we can understand fully the performance of our system. In particular, we examine the 10 000 training alerts and 500 testing alerts from the DARPA dataset to find out what type of alerts exist in these datasets. (The details of the selection of the DARPA training and testing datasets are discussed in Section 4.1.)

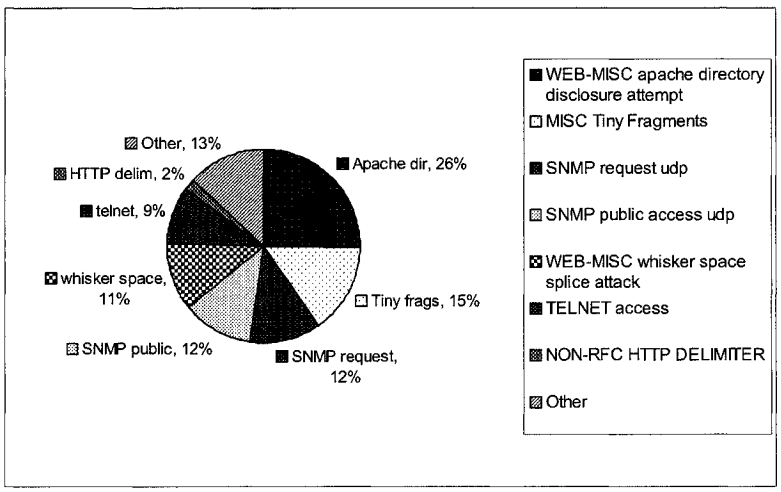

Figure 3.1: Percentage distribution of different types of alerts in the DARPA training dataset

As you can see from Figure 3.1, the composition of the training dataset is varied. The largest block of alerts is the set of "Apache directory disclosure attempt" alerts. Figure 3.1 shows that this type of alert forms the fourth largest block of alerts in the testing dataset. "Tiny [IP] Fragments" comprise a more significant proportion of alerts in the testing dataset than in the training dataset.

While it is clear that these two datasets are different, they both contain the same trends. For instance, we can see from these figures that "[*simple network management protocol (SNMP)*] request" and "SNMP public access" alerts comprise equal proportions of the two datasets.[1] It is rare that the percentage distribution of alert types on a network stays

[1] In fact it is likely that these two alert types usually occur simultaneously.

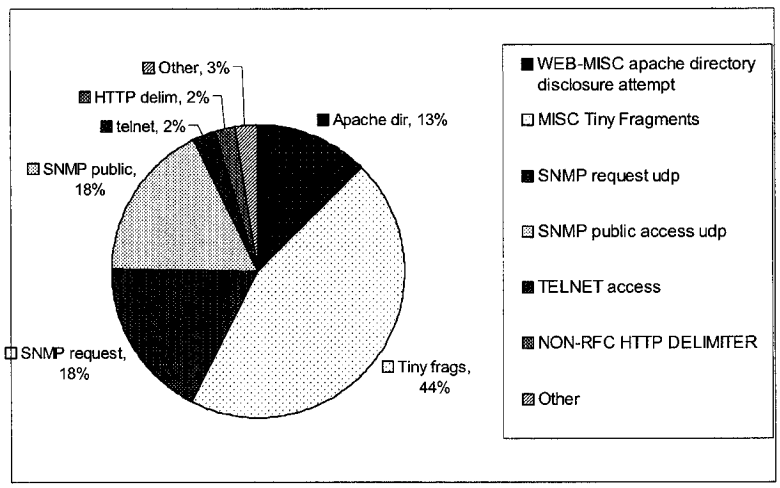

Figure 3.2: Percentage distribution of different types of alerts in the DARPA testing dataset

constant over time [3], so the composition of these datasets probably accurately models the real world.

3.1.3 Scaling Algorithms

In our system we scale the data linearly using maps based on the minimum and maximum possible values for each feature. Because we know these values for all of the features we encode, we can linearly transform values to the interval $[-1, 1]$. We know all of the minimum and maximum values for the features we encode because they are provided in the internet protocol specifications.[2] For instance, we know that TCP source and destination ports are 16-bit values, so we know that we must linearly transform values from the range $[0, 2^{16} - 1]$ to $[-1, 1]$ for those features. For TCP port p we perform this mapping using the formula $\frac{2}{2^{16}-1}p - 1$. We can generalize this formula for feature f and data item d_i as follows: If all values for f exist in the interval $[l^{(f)}, h^{(f)}]$, the scaled $d_i^{(f)}$ is $2\frac{d_i^{(f)} - l^{(f)}}{h^{(f)} - l^{(f)}} - 1$. We call this method "scaling to predefined feature ranges" in our experiments. This name is to contrast the difference between this method and other

[2]See Section 2.1 for details on the minimum and maximum values for the first stage features.

linear scaling methods we use, which scale based on automatically determined ranges.

As mentioned, we also experimented with scaling data using the method of determining a linear map from the training data then applying it to the training, testing, and evaluation data [71]. Specifically, for every feature in the training dataset we determine the range of values. For feature f, we call the highest value in the training dataset $h^{(f)}$ and the lowest value $l^{(f)}$. From these values we compute the linearly scaled value for feature f of a data item d_i as $2\frac{d_i^{(f)}-l^{(f)}}{h^{(f)}-l^{(f)}} - 1$. For the case when $h^{(f)} = l^{(f)}$, we compute f for d_i as $d_i^{(f)} - l^{(f)}$. You can see that with this computation, as long as all the input values are in the interval $[l^{(f)}, h^{(f)}]$ then the linearly mapped values will be in the interval $[-1, 1]$. If scaled values are outside the interval $[-1, 1]$ in the testing dataset they are left as-is. The ranges for the testing values should not be considered in advance to allow for a realistic representation of the testing data. The model must acknowledge that training data rarely perfectly predicts testing data, so it is expected that some testing values will lie outside the intervals $[l^{(f)}, h^{(f)}]$ determined in training.

Lin *et al* [80] describes why scaling is important: basically, if features are not scaled the importance of feature f can overwhelm other features if the value $h^{(f)} - l^{(f)}$ is significantly larger than that value for other features. Also, in neural networks computation errors can occur if the numerical values in the computations are too large.

We also wanted to investigate linearly scaling to the interval $[0, 1]$ using automatically determined ranges instead of $[-1, 1]$ because this range is sometimes recommended [80]. To do this we modified the values for feature f of data item d_i previously scaled to $[-1, 1]$ using automatically determined ranges, say $scaled([-1, 1], d_i^{(f)})$, and performed the following transformation to get $scaled([0, 1], d_i^{(f)})$:

$$scaled([0, 1], d_i^{(f)}) := \frac{1 + scaled([-1, 1], d_i^{(f)})}{2} \tag{3.1}$$

Lastly, we wanted to try a Gaussian method of scaling, as well, that relies on accurate — and Gaussian — distributions in the training data. To compute the Gaussian scaling for $d_i^{(f)}$, we first have to determine $\mu^{(f)}$ and $\sigma^{(f)}$. To do this, we used the mean and standard deviation calculations, respectively, of Matlab [68] on the training dataset. Once these

values were found, we could compute the new value for $d_i^{(f)}$ as $\frac{d_i^{(f)}-\mu^{(f)}}{\sigma^{(f)}}$. This gives the training data a new mean of 0 and a new standard deviation of 1. The values of $\mu^{(f)}$ and $\omega^{(f)}$ from the training set are used to scale the evaluation and testing sets.

For completeness we also discuss our results when experimenting with our system using unscaled data.

We present experiments with all of these scaling methods in Section 4.2.3.

3.2 First Correlation Stage

For the first stage of alert correlation in our system we use a set of features to represent an IP packet that has been flagged as an alert by an IDS, specifically the Snort IDS in our experiments. The feature set encourages similar IP packets to be clustered together. A clustering algorithm, using this set of encoded features, clusters IP packets received in a given window into individual steps of larger attacks. We give the details of how alerts are encoded as a set of features in Section 3.2.1.

For the purpose of our research we define an attack *step* to be one action in the attacker's greater plan. For instance, one step by an attacker might be running the security scanning tool `nmap` once against a network to discover what services are available. This step would likely generate many alerts. We define a step, in this case, to exclude the use of tools to attack particular services or the use of other reconnaisance tools by the same attacker. In the case of false positives we call all similar false positives generated by one source a step as well, for simplicity in explanations. False positives are not detected by our system *per se* and the correlation techniques do not treat possible false positives differently than real attacks.

We use a clustering algorithm to find correlations between similar alerts at the first stage of correlation. We evaluate three clustering algorithms for possible use at the first stage. We consider the autoassociator, as presented in Section 3.4.1, combined with the ClusterBarrier algorithm, described in Section 3.4.2, as one of the clustering algorithms. We show the implementation of the system with the autoassociator run

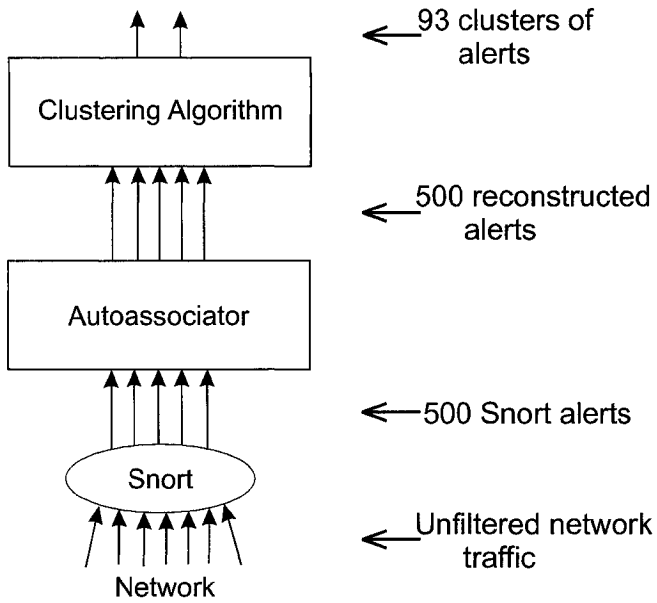

Figure 3.3: Example flow of alerts from the network through the first stage implemented with the autoassociator and single-link algorithm

on an example set of 500 alerts in Figure 3.3. Figure 3.3 shows how network traffic is monitored by the IDS, and how the output of the IDS is interpretted by the first stage of our alert correlation system to form clusters of alerts. We also consider using the EM algorithm and self-organizing maps at the first stage of correlation.[3]

Alert Window Selection

The first stage clusters a set of encoded alerts from a window of data. A window of data W can be a consecutive sequence of n alerts $a_1, \ldots, a_n$, where n is chosen ahead of time. A window of data can also be all of the alerts generated by the IDS in the interval of time $[t_1, t_2]$, the set of alerts $\{a_i : t_1 \leq a_i.t \leq t_2\}$, where $a_i.t$ is the generation time of alert a_i. The first stage outputs a set of clusters C such that $W = \bigcup_{c \in C} c$ and $c \cap d = \emptyset$

[3]We considered the well-known k-means clustering algorithm in our initial research [1] as well but we quickly concluded that its performance was significantly poorer than that of other cluster algorithms so we discontinued our consideration of that algorithm as a candidate for the first stage of correlation.

for all $c, d \in C$.

We separate the stream of data from the IDS into windows to make the results more manageable for the clustering algorithm at the first stage. Separating a stream of alerts into windows also allows a rough form of correlation based on generation time at the first stage as well. Generation time is a valuable attribute in alert correlation [9].

By separating a stream of alerts into windows, some sets of related alerts from the same attack step will certainly be split across windows of data. A set of related alerts can be split across windows of data because a single step of an attack can generate tens of thousands of alerts or last a very long time. We rely on the second stage of correlation to find relationships between clusters from different windows of data.

The first stage can either be applied to every consecutive window of alerts generated by the IDS, or it can be applied to particular windows selected by the IDS analyst. For the experiments in Chapter 4, we use apply the first stage to consecutive windows of 100 alerts. So we set n to be 100 in our experiments. We explain this choice of n in Section 4.1.1.

Training on Unlabelled Data

If the clustering algorithm being evaluated for the first stage can be trained on unlabelled alerts, the algorithm is provided with a set of 10 000 unlabelled alerts from the same collection of data. For instance, if the set of alerts to be correlated is from the incidents.org data, the set of 10 000 unlabelled alerts used for training is also extracted from the incidents.org data.

Our use of unlabelled data to train our unsupervised machine learning algorithm is supported by literature. Specifically in Duda, Hart and Stork [71] they suggest that it is possible to "train with large amounts of (less expensive) unlabelled data, and only then use supervision to label the groups found." This use of unlabelled data corresponds to our own use and thus gives us justification for our architectural decision.

3.2.1 Feature Construction

We encode 40 features to characterize each alert generated by Snort [37], the IDS we use in our testing. The features we use are taken directly from the Snort alerts. We did not need to include any information specific to Snort, so we argue that our selection of features could be extracted from any IDS rather than from only Snort.[4] For instance, we do not use the Snort attack priority level or the Snort labelling of the attack because this information cannot be found in the raw IP packet. We wanted to make our system as general as possible, so we showed how clustering with only protocol features can produce valid clusters.

Table 3.1: First stage features

Feature	Feature	Feature	Feature
portSrc	portDest	ipIsIcmpProtocol	ipIsIgmpProtocol
ipIsTcpProtocol	ipIsUdpProtocol	ipLen	ipDgmLen
ipId	ipTos	ipTtl	ipOptLsrr
ipPacketDefrag	ipReserveBit	ipMiniFrag	ipFragOffset
ipFragSize	icmpCode	icmpId	icmpSeq
icmpType	tcpFlag1	tcpFlag2	tcpFlagUrg
tcpFlagAck	tcpFlagPsh	tcpFlagRst	tcpFlagSyn
tcpFlagFin	tcpLen	tcpWinNum	tcpUrgPtr
tcpOptMss	tcpOptNopCount	tcpOptSackOk	tcpOptTs1
tcpOptTs2	tcpOptWs	tcpHeaderTrunc	udpLen

We did not include the IP source address or IP destination address protocol fields in the list of clustering features because we noticed that these features tended to impede correct clustering. We also reason that if we wish to cluster a number of packets together that make up a distributed denial of service or similar attack, the IP address fields could impede correct clustering since source IP address is easily forged[5] for certain types of attacks, and the destination IP address can be masked by attacking many computers at once even if only one computer is the target.[6]

[4]We do not use the IDMEF standard [54] because the set of mandatory features provided in this standard is insufficient for the type of correlation we perform, so vendor-specific alert processing is always necessary.

[5]This is sometimes called IP spoofing.

[6]We note, however, that we do use IP source and destination addresses in our second stage of

We included the TCP source and destination port features because we find they increase performance for clustering types of attacks on particular TCP services. We do not use the TCP sequence and TCP acknowledgement numbers at this stage because we found that they hinder performance in experiments.

We tested, evaluated, and considered the other features individually, but we decided to use all of the features listed in Table 3.1 because they all helped produce correct clusters. In the end, we encode 40 features that we pass to the clustering algorithm for clustering.

Formatting

All of the 40 features we selected are intuitively encoded as numerical data. For instance, if the TCP ACK flag is set in the TCP header, the value of the `tcpFlagAck` feature is set to 1, and if the flag is clear or there is no TCP header in the IP packet, the `tcpFlagAck` feature is set to 0.

Some features of note are the `ipIsIcmpProtocol`, `ipIsIgmpProtocol`, `ipIsTcpProtocol`, and `ipIsUdpProtocol` binary features. If an encoded data item has the value 1 for the `ipIsTcpProtocol` feature, this indicates that the Snort alert represented by the encoded data item contains a TCP header and payload. A Snort alert for a non-TCP IP packet will have `ipIsTcpProtocol` feature set to 0. Only one of the `ipIs*Protocol` features will be set for a given data item, since IP packets can contain at most one upper-layer protocol.

Formatting Example

To illustrate precisely how the transformation is made from a textual alert to a data item, and what information is encoded, we present an example. (Although the encoded data items can be scaled as well, we present an unscaled data item here for readability.) The

correlation. Performance gains were seen at the second stage of correlation when including these features because they are valuable in detecting an attack source for many types of attacks. We found that allowing alerts with dissimilar IP addresses to be clustered together at the first stage adequately solved the problem of correlating denial of service attacks.

following Snort-generated alert was parsed to generate a tuple of numbers, where each number corresponds to a predetermined feature. The alert below was generated because an entity tried to access the UNIX `telnet` service. It is possible that this alert could have been generated by a valid user, but it is also possible that this alert could have been generated by an attacker trying to penetrate the system. If this second hypothesis were true, it is likely that an analyst would find many "TELNET login incorrect" errors close to this one targetting the same set of machines.

```
[**] [1:718:7] TELNET login incorrect [**]
04/05-09:38:26.936288 172.16.114.50:23 -> 172.16.114.168:10332
TCP TTL:64 TOS:0x10 ID:2014 IpLen:20 DgmLen:66 DF
**AP*** Seq:0x801D2FDC Ack:0x6EDB5C7F Win:0x7C00 TcpLen:20
```

We present Table 3.2, which contains the encoded features of this alert. As an example of how to read Table 3.2, in the data below the number 23 is the TCP source port (`portSrc`) given in the Snort alert.

As you can see from the Snort alert, the TCP header in this packet has only the ACK and PSH TCP flags set. (The string `***AP***` indicates this.) This information is encoded in the `tcpFlag*` group of features. `tcpFlagAck` and `tcpFlagPsh` are set to 1 and the rest of the flags are set to 0.

The `portSrc` and `portDest` features represent the TCP source and destination ports indicated in the Snort alert. The values of the `portSrc` and `portDest` features are taken directly from the alert, as you can see above. The values for `portSrc` and `portDest` are 23 and 10332 respectively. The TCP source and destination port features are important in finding valuable correlations because they are sometimes the only obvious indicator as to which higher-layer protocol the alert relates.

3.3 Second Correlation Stage

With the second stage in our correlation system we aim to correlate the clusters produced by the first correlation stage to form *super-clusters*, or clusters of clusters of alerts. We

Table 3.2: An encoded telnet alert

Feature	Value	Feature	Value
portSrc	23	portDest	10332
ipIsIcmpProtocol	0	ipIsIgmpProtocol	0
ipIsTcpProtocol	1	ipIsUdpProtocol	0
ipLen	20	ipDgmLen	66
ipId	2014	ipTos	16
ipTtl	64	ipOptLsrr	0
ipPacketDefrag	1	ipReserveBit	0
ipMiniFrag	0	ipFragOffset	0
ipFragSize	0	icmpCode	0
icmpId	0	icmpSeq	0
icmpType	0	tcpFlag1	0
tcpFlag2	0	tcpFlagUrg	0
tcpFlagAck	1	tcpFlagPsh	1
tcpFlagRst	0	tcpFlagSyn	0
tcpFlagFin	0	tcpLen	20
tcpWinNum	31744	tcpUrgPtr	0
tcpOptMss	0	tcpOptNopCount	0
tcpOptSackOk	0	tcpOptTs1	0
tcpOptTs2	0	tcpOptWs	0
tcpHeaderTrunc	0	udpLen	0

hope to link clusters from the first stage that are related, but that do not form one discrete cluster after the first stage. Effectively, we hope to be able to correlate different steps of an attack together by representing each step such that it can be correctly clustered with other steps from the same attack.

The motivation for this second stage of correlation comes from Valdes and Skinner [52]. In their approach they use statistical methods primarily based on the type of the alert and the source and destination IP addresses of the alert. They correlate alerts using three distinct stages in their system. The first stage threads the alerts of a given sensor so that each logical step of an attack at a given sensor is not represented multiple times. The second stage of their system combines the output of multiple IDS sensors to fuse the data into one stream. After this second stage each step of an attack should be represented only once in their system. For the final stage of their system, they try to correlate their already-formed groups of alerts based on which groups are part of the same greater attack. At this stage they use source IP addresses to determine attacker origins and destination IP addresses to determine targets. They also use predefined matrices of values that represent how likely it is that two types of alerts are related, if seen in succession.

In our system we combined the first two stages of the Valdes and Skinner system into one stage, namely the first stage in our system. Our system is fairly successful at threading a set of alerts, and we believe — without proof, but with strong indication, since our methods are sensor independent — that our system could handle multi-sensor threading. (We have not proven this latter claim because multi-sensor data is not available to us.)

Adding a second level of clustering effectively mimics the intuitive approach by Valdes and Skinner of finding attack step relationships after the threading and multi-sensor correlation has been done. We should note, though, that both of the other systems we model our approach on, the systems of Dain *et al* [51] and Julisch *et al* [3], do not use a multi-stage correlation approach. The difference is that their systems are both based on supervised machine learning, whereas ours is unsupervised, so the structures in the correlations we produce have to be imposed rather than learned.

We implement the second stage of our system using a clustering algorithm, as with the

first stage of our system, but with different features to reflect what we want to learn at the second stage. We show how these features are constructed from the output of the first stage in Section 3.3.1.

The second stage finds super-clusters in a consecutive sequence of m windows of alerts, $W_1, \ldots, W_m$. Each window of alerts, W_i, is represented by a number of clusters of alerts, C_i, for the second stage algorithm. The clusters of alerts C_i were generated by applying the first stage to window W_i. The second stage input is the set of clusters $\bigcup_{i=1}^{m} C_i$, each cluster individually encoded by the set of features in Section 3.3.1. The number of data items input to the second stage clustering algorithm is $\sum_{i=1}^{m} |C_i|$. As you can see, by combining the clusters of m windows of alerts, the second stage clustering algorithm can find correlations between any alerts within the m windows.

In our second stage clustering algorithm and parameter selection experiments in Section 4.3, we combine the first stage output of 5 consecutive windows of alerts. The value m is 5 in our overall performance tests in Section 4.4 as well. The choice of m is explained in Section 4.1.1.

3.3.1 Feature Construction

For the second correlation stage, we take the output clusters of the first stage and represent them each as new data items, each encoded using a new set of features. Feature construction for the data ultimately determines what is learned in machine learning problems. Therefore, we intentionally constructed our features such that a cluster of alerts representing a single step of an attack will be similar to another cluster of alerts from the same attack. We construct our features such that the converse is also true, by necessity.

We take our motivation for feature construction at this stage from the work of Dain *et al* [51]. Dain *et al* describe a number of features used in their supervised learning system that we have included in our system.

Before describing the features in detail, we present our method for determining the similarity of IP addresses, since it differs slightly from the Dain *et al* method. A measure of closeness between two IP addresses is used in the Dain *et al* system that is applied to source and destination IP addresses of alerts. In their system closeness is defined as the maximal number of most-significant bits shared between two IP addresses encoded as 32-bit integers[7], which they call r. So, for instance, in their system if one IP address is 192.168.2.17 and another IP address is 192.168.2.16, these would be deemed very similar ($r = 31$), whereas the IP address 192.168.1.16 would be considered less similar to the previous two ($r = 22$). This measure of closeness is valid because pairs of IP addresses with high r values are likely to be close within the network [6].

Because our system is a clustering system where the features of all data are compared at once for similarity, we cannot implement a measure to compare pairs of IP addresses. Accordingly, we implement a feature that makes a set of IP addresses similar if and only if they would be similar in the Dain *et al* system. In our system, we represent the similarity of a set of IP addresses by representing them as the IP addresses of the group with the unshared least-significant bits masked out. So, using the previous examples, the group of IP addresses (192.168.2.16, 192.168.2.17) would be represented with the single IP address 192.168.2.16, and the group of IP addresses (192.168.2.16, 192.168.2.17, 192.168.1.16) would be represented as 192.168.0.0.

We now discuss the features used by Dain *et al*, and the analogous features in our system, in the following list. (Each of the items of text in the following list is preceded by the name of the feature in our system.)

1. `ipSrcAddrCommonPart` — Attackers often perpetrate an attack from a single host, or from a single IP subnet. So Dain and Cunningham included a feature to indicate similarity between the source IP addresses of two alerts. In our system, we included a feature to indicate the shared part of the most-significant bits of a group of IP addresses from a cluster.

2. `ipDestAddrCommonPart` — Dain and Cunningham asserted that attackers often

[7]IP address $w.x.y.z$ is encoded as the 32-bit integer $256^3w + 256^2x + 256y + z$. This is the same encoding formula presented in Section 2.1.

target a single host or subnet in their attacks. So they included a feature indicating the similarity of two destination IP addresses. For our system we constructed a feature for destination IP addresses analogous to the feature for source IP addresses.

3. `ipSrcAddrCommonBits` — Dain and Cunningham assert, "An attack scenario may contain components with spoofed source IP addresses while other components of the attack may use the attackers real source IP." As such, they included a feature to indicate the minimal r value found between the source IP of an alert and the source IPs of a group of alerts. We include the same feature in our system, but computed such that the feature indicates the minimal value of r between any pair of source IP addresses in a cluster.

4. `avgTimeSig` and `varTimeSig` — Dain and Cunningham have features to indicate the similarity in time between when two alerts were generated. In our system we have features to indicate the average time when a group of alerts was generated, and the standard deviation (named `varTimeSig`) for the times when a group of alerts were generated.

5. `avgReconsErr`[8] — Dain and Cunningham have features to indicate whether a new alert is of the same type as the most recent alerts of already established groups. In our system, we have a feature to indicate the average reconstruction error of a group of alerts. As we establish in experiments (see Section B.2), the reconstruction errors of alerts tend to correspond directly with their type.

There are a few features from the Dain and Cunningham work that we were not able to represent in our system. In their system, and in other correlation systems, researchers include a feature to indicate the similarity of a source IP address and a destination IP address. They suggest that this might be useful in classification or clustering because the source IP address of a new alert can be the destination of an old one, if an IP address in your network has been compromised in a previous part of an attack. When a compromised computer is used to obscure the true source of an attack or it is used for its enhanced access priviledges, it is called a *stepping stone*. Because of the structure of the

[8]Note: this feature is not available if the clustering algorithm from our first stage wasn't based on the autoassociator.

Dain *et al* data, the packet captures of a DEFCON hacking contest [58], it is very unlikely that this feature was used how they designed it to be used. Attackers did not use the target machines as stepping stones in the contest because the contest was not structured to award points for hiding an attack. We were able to design a feature to encode stepping stone information, but we chose not to implement it for the non-operational system we created.

Another set of features Dain *et al* included in their system related to a technical problem of the RealSecure IDS that they used in their experiments. Apparently this IDS sometimes switches the source and destination IP address in alerts, so they had features to account for the switch. In our system, we used only Snort to experiment with and we did not see any evidence of this, so we did not feel there was a problem to overcome by creating new features.

We also experimented with some other features that Dain *et al* did not consider. We list them below in the same format as the previous list.

1. `modePortSrc` and `modePortDest` — We found that using the TCP source and destination ports was sometimes valuable in determining the grouping of a set of alerts. We also found that when the ports were useful, there was almost always a particular source or destination port that was more common than the rest. Accordingly, we created two features: one feature to encode the most common TCP source port in a cluster of alerts and a feature to encode the most common destination port.

2. `avgSeqNumDiff` — We also found that TCP sequence numbers and TCP acknowledgement numbers are sometimes good predictors of the use of a hacker tool. Often there are obvious patterns in a group of sequence or acknowledgement numbers, and this pattern can usually be encoded by taking the difference of the two numbers. Doing this also helps in the specific situation where there is an explicit relationship between these two sets of numbers within a cluster. We encoded the average difference between these two numbers for each cluster.

Formatting Example

We now present an example of how a cluster produced by the first stage of our system can be formatted as a data item in our second stage. Table 3.3 contains the following two alerts encoded as described in Section 3.2.1.

```
[**] [1:618:5] SCAN Squid Proxy attempt [**]
11/14-16:06:21.366507 206.48.61.139:4006 -> 170.129.23.239:3128
TCP TTL:114 TOS:0x0 ID:24710 IpLen:20 DgmLen:48 DF
*****S* Seq:  0x13D84F7 Ack:  0x0 Win:  0x2000 TcpLen:  28
TCP Options (4) => MSS: 536 NOP NOP SackOK

[**] [1:618:5] SCAN Squid Proxy attempt [**]
11/14-16:06:24.316507 206.48.61.139:4006 -> 170.129.23.239:3128
TCP TTL:114 TOS:0x0 ID:26758 IpLen:20 DgmLen:48 DF
*****S* Seq:  0x13D84F7 Ack:  0x0 Win:  0x2000 TcpLen:  28
TCP Options (4) => MSS: 536 NOP NOP SackOK
```

Then, Table 3.4 contains the features constructed using the results of the first stage and the values in Table 3.3.

3.4 Correlation and Clustering Algorithms

We consider three clustering algorithms for use at the first and second stages of correlation in our system. Specifically, we try the autoassociator with the ClusterBarrier algorithm, presented in Section 3.4.1 and Section 3.4.2, the EM algorithm, discussed in Section 2.4.3, and self-organizing maps, in Section 2.4.3. We experiment with various parameters and traits of these algorithms in Chapter 4 and Appendix B.

We discuss desirable traits of the clustering algorithms we consider in Section 3.4.3. We also present the simple NaiveCorrelator algorithm in Section 3.4.4 against which we compare our system in the overall performance results of Section 4.4.

Table 3.3: Two encoded scan alerts from the first stage

Feature	Alert 1	Alert 2	Feature	Alert 1	Alert 2
portSrc	4006	4006	portDest	3128	3128
ipIsIcmpProtocol	0	0	ipIsIgmpProtocol	0	0
ipIsTcpProtocol	1	1	ipIsUdpProtocol	0	0
ipLen	20	20	ipDgmLen	48	48
ipId	24710	26758	ipTos	0	0
ipTtl	114	114	ipOptLsrr	0	0
ipPacketDefrag	1	1	ipReserveBit	0	0
ipMiniFrag	0	0	ipFragOffset	0	0
ipFragSize	0	0	icmpCode	0	0
icmpId	0	0	icmpSeq	0	0
icmpType	0	0	tcpFlag1	0	0
tcpFlag2	0	0	tcpFlagUrg	0	0
tcpFlagAck	0	0	tcpFlagPsh	0	0
tcpFlagRst	0	0	tcpFlagSyn	1	1
tcpFlagFin	0	0	tcpLen	28	28
tcpWinNum	8192	8192	tcpUrgPtr	0	0
tcpOptMss	536	536	tcpOptNopCount	2	2
tcpOptSackOk	0	0	tcpOptTs1	0	0
tcpOptTs2	0	0	tcpOptWs	0	0
tcpHeaderTrunc	0	0	udpLen	0	0

Table 3.4: A cluster encoded for the second stage

Feature	Value	Feature	Value
numAlerts	2	ipSrcAddrCommonPart	3459267979
ipSrcAddrCommonBits	32	ipDestAddrCommonPart	2860586991
ipDestAddrCommonBits	32	modePortSrc	4006
modePortDest	3128	avgIpHdrLen	20.0
avgPayloadLen	48.0	avgReconsErr	6.7834
avgSeqNumDiff	20808951.0	avgTimeSig	974228782.5
varTimeSig	2.12132034356	avgTcpFlagsSet	1.0

3.4.1 The Autoassociator

We consider the autoassociator for clustering at each stage of correlation. We first considered this neural network algorithm because it is very flexible and because we found it performed well in initial experiments. Another reason we consider this algorithm is that the neural network can be trained with unlabelled training data, which makes the results between successive runs of the algorithm on different windows of alerts comparable, due to the stability in the reconstruction error values. This trait is important for the first stage of correlation so that the results can be used in the second stage. (We discuss this topic more later in this section.) The final reason we consider the autoassociator is because the autoassociator, used with the CLUSTERBARRIER algorithm in Section 3.4.2, does not require the number of output clusters to be known beforehand.

To train the autoassociator we selected 10 000 unlabelled alerts and trained the autoassociator[9] on this data for a number of epochs, using a set number of hidden units in the construction of the network. We explore these parameters in the experiments in Section 4.2.1 and Section 4.2.2. We also experiment with the presence of the training set of 10 000 unlabelled alerts in Section B.1.

After the autoassociator was trained we simulated our set of evaluation alerts on the network to obtain the reconstruction error values for each alert. We clustered the evaluation data using these reconstruction errors and the CLUSTERBARRIER algorithm of our own creation.[10] The details of this algorithm are presented in Section 3.4.2. The use of the reconstruction error formula creates the problem of a 40-to-1 mapping that causes some errors in clustering. We discuss the effects of this mapping below. We formulated an experiment to learn more about the mapping problem in Section B.3.

[9]The autoassociator was implemented in Matlab for our tests. We used the backpropagation algorithm and the gradient descent learning rule to train the autoassociator.

[10]We considered substituting other clustering algorithms as the algorithm to interpret autoassociator output in other research [2], but we found in this work that the cluster barrier algorithm performs as well as or better than combining another clustering algorithms with the autoassociator.

Training the Autoassociator

Machine learning is used in this thesis to cluster numerically similar IDS alerts. We will show how the autoassociator can be used for exactly this task. A simple clustering algorithm is required to interpret the neural network output and to form discrete clusters, which we discuss in the Section 3.4.2. But the autoassociator is used in essentially the same way as it is described in Section 2.4.2, except that the neural network is trained on unlabelled data from multiple classes or clusters, rather than training on only one class of data as in Japkowicz [4].

In our task we use the autoassociator for clustering rather than for binary classification. Where Japkowicz found a threshold in the reconstruction error values to separate two classes of data, we group the reconstruction error values to form clusters. The difference in reconstruction error of two given data is positively correlated with the general numerical difference of those data. In our task of alert correlation, if the numerical representation of two alerts is similar, the autoassociator will produce similar reconstruction errors. The converse is often true, but not always true: numerically dissimilar alerts often produce dissimilar reconstruction errors, but not always. The reason this is the case is that the reconstruction error formula maps the 40 autoassociator outputs to the single-dimensional range of error reconstruction values. Because of this 40-to-1 mapping, there is a lot of information lost, so numerically different autoassociator outputs can be mapped to the same reconstruction error. We explore this problem further below.

As an example of the correlation of reconstruction error values, the last two Snort alerts listed below are very similar and the first is dissimilar from the last two. Since the last two alerts are numerically similar, they have very similar reconstruction errors (which are the same in this particular case: 3.1920 for both alerts, after 500 epochs of training), but the last two have substantially different reconstruction errors from the first Snort alert below (which had reconstruction error 3.3388 in the same experiment).

```
[**] [1:716:10] TELNET access [**]
04/05-09:35:53.754855 172.16.112.50:23 -> 172.16.114.148:7298
TCP TTL:255 TOS:0x0 ID:28873 IpLen:20 DgmLen:55 DF
**AP*** Seq:0x367323BE Ack:0xF384C1BE Win:0x2238 TcpLen:20
```

```
[**] [1:1244:13] WEB-IIS ISAPI .idq attempt [**]
04/05-09:37:05.489493 172.16.117.103:8307 -> 208.160.10.123:80
TCP TTL:64 TOS:0x0 ID:4996 IpLen:20 DgmLen:549 DF
**AP*** Seq:0x336579CE Ack:0x7B33D444 Win:0x7D78 TcpLen:20

[**] [1:1245:10] WEB-IIS ISAPI .idq access [**]
04/05-09:37:05.489493 172.16.117.103:8307 -> 208.160.10.123:80
TCP TTL:64 TOS:0x0 ID:4996 IpLen:20 DgmLen:549 DF
**AP*** Seq:0x336579CE Ack:0x7B33D444 Win:0x7D78 TcpLen:20
```

From the list of reconstruction errors of a set of Snort alerts, we can use a simple algorithm to form discrete clusters of Snort alerts whose numerical representations are similar. The clustering step is necessary because the autoassociator does not explicitly form clusters. We discuss how this algorithm works in Section 3.4.2.

Autoassociator Training Parameters

As discussed in Section 2.4.2, the hidden units parameter in neural networks defines the learning capacity of the network. The hidden unit layer of a multi-layer perceptron neural network is responsible for allowing the learning of non-linear concepts. There exists a (possibly non-unique) optimal number of hidden units in a neural network for a given training set. By tuning this neural network parameter, one can approximate the optimal number of hidden units, and thus attain better performance in the particular domain.

The epochs parameter in neural networks defines the number of iterations of the neural network training algorithm, as seen in Section 2.4.2. As the neural network is trained for more epochs, the network is more capable of recognizing the particular data on which it is being trained, but with the caveat that overfitting can occur.

We consider the hidden units and epochs parameters the most important in their ability to affect the success of the autoassociator as a clustering algorithm, so we experiment with them in Section 4.2.2.

Feature Mapping Problem

The reconstruction error formula (Equation 2.12 in Section 2.4.2) computes a single real value from the 40 autoassociator outputs, causing a 40-to-1 mapping. It is necessary for our system to create discrete clusters from the otherwise difficult-to-interpret 40-dimensional output of the autoassociator, which motivates our use of this 40-to-1 mapping formula. We considered other clustering algorithms in place of this formula as well in Section 4.2. We considered the reconstruction error formula as the method for performing the 40-to-1 mapping because it was used in Japkowicz [4], where the formula was used to generate ROC graphs [41, 42] in a binary classification problem.

The problem lies in the fact that different 40 autoassociator outputs can be mapped to the same real value using the formula. Our cluster barrier algorithm (Section 3.4.2) then sees the different autoassociator outputs as the same and clusters the items together.

We explore the extent of this problem in Section B.3 and discuss possible solutions.

Reconstruction Error Stability

Stability is an important trait of the autoassociator trained with 10 000 alerts. If the correlation system is run against multiple consecutive windows of alerts, it is important that the results between these windows will be comparable. We must define stability in the context of reconstruction error values changing between experiments. Stability for a particular alert or data item means that for a given training set that the reconstruction error value of the alert does not vary signficantly regardless of the composition of the evaluation set that the alert is part of. What this means practically is that the reconstruction error of an alert should be mostly independent of the dataset that contains it.

We show the stability of the reconstruction error values produced by our system in an experiment in Section B.2.

3.4.2 Cluster Barrier Algorithm

The goal of the ClusterBarrier clustering algorithm is to create clusters with well-defined boundaries from a list of reconstruction error values. The algorithm is used to form discrete clusters from the output of the autoassociator. The input to the algorithm is a list of reconstruction errors computed from the autoassociator outputs. Each alert simulated by the autoassociator has a corresponding reconstruction error value. The list of reconstruction errors computed from the autoassociator outputs is a list of real numbers with no obvious boundaries, but with intuitive clusters of values. The reconstruction error formula maps a 40-dimensional input vector to a 1-dimensional output value, to explain where the 40-to-1 mapping occurs in relation to the clustering algorithm. The clustering algorithm functions on a list of 1-dimensional inputs.

To achieve the goal of creating well-formed clusters, we found that a heuristic approach worked well. We decided that any cohesive gaps in the list of sorted reconstruction error values would indicate a boundary between clusters, so we used that idea.

Algorithm 2 ClusterBarrier(S, b)

Input: An unordered, non-empty set of alerts, S, with reconstruction errors computed by the autoassociator.

Input: A value, b, for the minimal distance between the reconstruction error values of a pair of alerts where a cluster separator can be placed.

Output: A set, C, of clusters (sets) of the alerts in S.

```
1: A ← Sort alerts in S using index reconsErr (reconstruction error). // Note: Array
   A is zero-based.
2: C ← ∅
3: c ← {A[0]}
4: for i ← 1 to |A| − 1 do
5:     if |A[i].reconsErr − A[i − 1].reconsErr| ≤ b then
6:         c ← c ∪ {A[i]}
7:     else
8:         C ← C ∪ {c}
9:         c ← {A[i]}
10: C ← C ∪ {c}
11: return C
```

We form discrete clusters such that every cluster obeys this rule: When you sort the

reconstruction errors of the data items, the difference between the reconstruction errors of any pair of items adjacent in the sorted list can be at most b.[11] We present the CLUSTERBARRIER algorithm as Algorithm 2.

We developed this heuristic from the data, and we find that it has worked well for our purposes. We show this heuristic at work in Figure 3.4. The sorted reconstruction error values of a sequence of 100 alerts[12] from the incidents.org dataset are plotted. The diamonds in the graph correspond to reconstruction error values of the alerts and the bars correspond to the difference in reconstruction error values between two consecutive reconstruction error values in the sorted list. You can see at points where there is a large vertical gap between plotted diamonds that the difference in reconstruction error values is greater than 0.0025. According to this graph there are 9 places where there is a gap in reconstruction error values greater than 0.0025, which means that our clustering algorithm would produce 10 discrete clusters from this data.

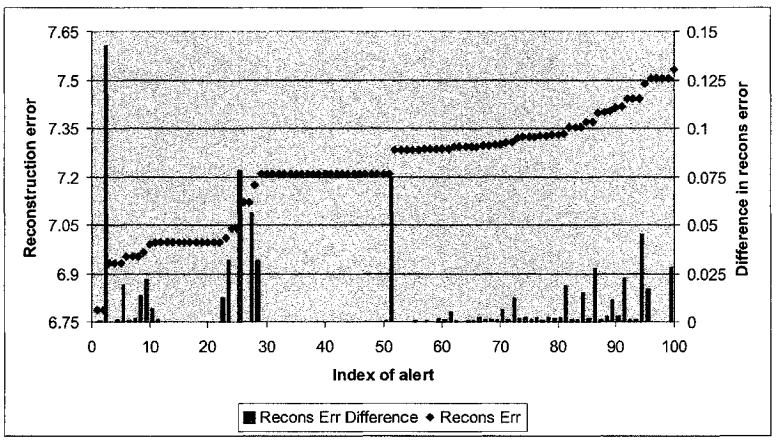

Figure 3.4: Sorted reconstruction error values of alerts in test clusters from incidents.org dataset

The heuristic we present is formally known as a *single-link clustering algorithm* (see Section 2.4.3 for details) applied to one-dimensional data — namely the reconstruction errors of the data. The clustering algorithm is known as a single-link clustering algorithm

[11] We find that a good value for b is 0.0025 in experiments in Section 4.2.

[12] The value of n, from Section 3.2, chosen in Section 4.1.1.

because if the distance between any two data in a set is less than the threshold then those two data will appear in the same cluster. We also call this algorithm the *cluster barrier algorithm* because of how we have conceptualized the algorithm.

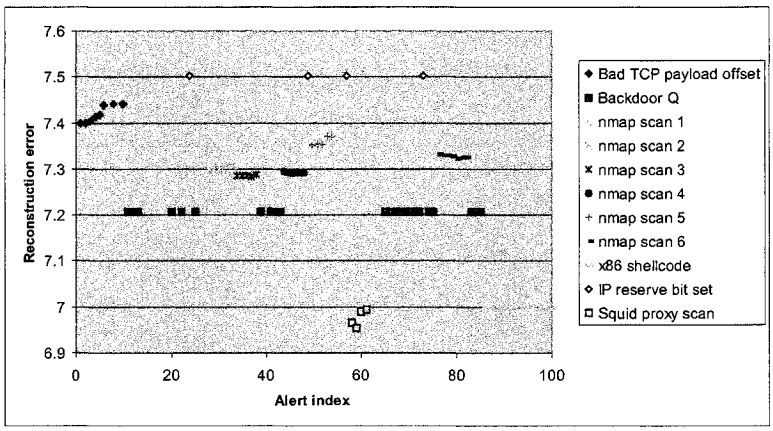

Figure 3.5: Reconstruction error of various clustered alerts from incidents.org dataset

We demonstrate how the algorithm works in practice with the autoassociator and the incidents.org dataset. To show this, we plot the index of a set of alerts against the reconstruction errors for those alerts in Figure 3.5. The alerts in this figure are the same 100 alerts as in Figure 3.4, except that they are unsorted in this graph. The legend in Figure 3.5 shows the list of clusters if the alerts were clustered by hand by an analyst, with every alert assigned to a cluster. As you can see, for some by-hand clusters, such as the "Backdoor Q" alerts, all the alerts in the cluster have the same reconstruction error value despite being spread across the whole dataset. This situation is positive because it means that the real cluster formed for this group of alerts will be the same as the by-hand cluster. But some by-hand clusters of alerts, such as the "x86 shellcode" alerts, will inevitably be split over multiple real clusters, since the reconstruction error values of the alerts of this type vary more significantly. Other problems occur when alerts of different by-hand clusters are clustered together, such as "nmap scan" clusters 1 through 4.

We present Figure 3.5 to illustrate how the reconstruction error values are used to cluster alerts and what sorts of errors can crop up with this clustering method.

We experimented with the size of the barrier between clusters before settling on the value 0.0025. We show this experimentation in Section 4.2.3.

3.4.3 Clustering Algorithm Traits

There are certain traits or characteristics of each candidate clustering algorithm that we desire but do not require. These traits are present in some clustering algorithms but not in others. We discuss why they are useful in the problem of alert correlation below. We also discuss the three candidate clustering algorithms in terms of the traits.

Number of Clusters

The first trait we desire is the ability for the clustering algorithm to determine automatically the number of clusters to output. This trait is desirable at both the first and second stages we do not know beforehand how many clusters or super-clusters of alerts should be formed. These values should be determined from the data, since the number of attack steps and number of attacks in a dataset is inherent to the data. Using heuristics to determine the number of attack steps or attacks can only approximate the number of clusters to form, so we do not explore heuristics of this sort in this thesis.

The autoassociator, employed with the CLUSTERBARRIER algorithm, is able to choose the number of clusters automatically. The number of clusters formed is partially controlled by the CLUSTERBARRIER algorithm parameter b, but once this parameter is set, the number of clusters is essentially chosen from the data.

When constructing SOMs, it is necessary to choose the dimensions of the map.[13] The dimensions of the map effectively define the number of clusters that can be formed. For example, a 4×6 map will have $4 \cdot 6 = 24$ neurons, which means that at most 24 clusters can be formed by this map. SOMs can have neurons with empty Voronoi sets, that is without any evaluation data items close to the neurons. This means that while

[13]The dimensionality of the map can also be chosen, but we fix it at a typical two dimensions for this thesis.

the maximum number of clusters in the previous example is 24, the actual number of nonempty clusters can still be as low as 1.

The EM algorithm explicitly requires the number of clusters to be chosen before clustering operation. But, as with self-organizing maps, some clusters can be empty, so the number of clusters chosen is actually the maximum number of clusters, and the actual number of nonempty clusters is sometimes much lower.

Adaptability to Data Stream

A desirable trait of the clustering algorithm we choose for the first stage of correlation is the ability for the algorithm to adapt to the stream of alerts produced by the IDS automatically. Adapting to the stream of alerts means that the correlations produced by the clustering algorithm are positively influenced by alerts previously seen by the system. If the clustering algorithm can use previous unlabelled data from the stream of alerts, this can reduce the maintenance requirements on the operator of the alert correlation system since changes in the composition of alerts reported by the network do not have to be manually encoded by an expert.

Adapting to a stream of alerts effectively means being able to train on previously seen alerts in our problem. We only have unlabelled data available to us because labelling data is impractical, so the clustering algorithm must be able to be trained on unlabelled data. The autoassociator and self-organizing maps can both be trained on unlabelled data, but the EM algorithm cannot be trained on unlabelled data. The EM algorithm can only be trained on labelled data.

Canonical First Stage Output

The second stage of correlation is able to use canonical rankings of alerts in its operation. If the first stage clustering algorithm can assign a number to an alert such that the same alert is assigned roughly the same number across windows of data, then the second stage of clustering can use this to identify clusters that are composed of similar alerts.

The autoassociator is able to assign numbers to alerts in this way roughly using the reconstruction errors of alerts. As we explore in Section B.2, we find that training the autoassociator on a large set of unlabelled training influences this ability of the autoassociator. The EM algorithm and self-organizing maps do not produce canonical numberings of alerts.

3.4.4 Naive Correlation Algorithm

We needed to compare the performance of our alert correlation system to the performance of another correlation system. We considered some commercial system and research systems, but we required a system that has goals comparable to our own. This requirement excluded many of the systems discussed in Section 2.3. In particular the correlation systems based on machine learning by Dain *et al* [51] and Julisch *et al* [3] use labelled training data that we do not have access to because the labelled data is not public. Another concern is that if we were able to implement a research system ourselves, we would not be assured that the system was implemented correctly, if the system is complex. We decided to implement a simple algorithm, which we called NAIVECORRELATOR, based on the correlation descriptions of Northcutt [9] and the web-based ACID alert console [61].

We created our own naive correlation system[14] that we can use as a benchmark for our performance. We can then compare the performance of our system to the performance of the simple system to test how well our system performs. We present the NAIVECORRELATOR algorithm that we have created as Algorithm 3.

The method of correlation that Northcutt advocates, and the NAIVECORRELATOR algorithm we create, is based on correlation using three primary variables: the type of alert to be correlated, the time that the alert was generated, and the source IP address of the alert. This information alone allows us to form discrete clusters similar to those formed by the unsupervised machine learning-based correlation system in this thesis.

To form clusters we first sort the alerts based on the Snort alert type, the source IP address and when Snort generated the alert. We create discrete clusters such that the

[14]We call this algorithm "naive" because it is so simple and unoptimized.

Algorithm 3 NAIVECORRELATOR(S)

Input: An unordered, non-empty set of alerts, S.
Output: A set, C, of clusters (sets) of the alerts in S.

```
1: A ← Sort alerts in S using indices type (Snort alert type), ipSrc (source IP address),
   and time (alert generation time). // Note: Array A is zero-based.
2: C ← ∅
3: c ← {A[0]}
4: for i ← 1 to |A| − 1 do
5:     if A[i].type = A[i − 1].type and
         A[i].ipSrc = A[i − 1].ipSrc and
         |A[i].time − A[i − 1].time| ≤ 24 hours then
6:         c ← c ∪ {A[i]}
7:     else
8:         C ← C ∪ {c}
9:         c ← {A[i]}
10: C ← C ∪ {c}
11: return C
```

first two attributes are equal and the difference between consecutive generation times is at most 24 hours.

The type of correlation system presented as the NAIVECORRELATOR algorithm is simple, but it adequately forms a base line to which we can compare our full alert correlation system. We compare the results of the NAIVECORRELATOR algorithm to the results of our full system in Section 4.4.

Chapter 4

Results Analysis

We present a number of experiments here, most of which are motivated by details of our model described in Chapter 3. The actual experiments and results evaluation are in Section 4.2 and Section 4.3. We discuss how we evaluate the experiments in Section 4.1, and the type of errors that we encounter in Section 4.1.3. We explore the overall performance of the system in Section 4.4.

4.1 Alert Data and Evaluation

To evaluate the performance of our system we created a gold standard to influence the type of results we want to see produced automatically by our system. The *gold standard* is three sets of data that were clustered by hand by us before we experimented with our system to show how we would like our system to correlate the same data. When creating the gold standard we tried to be faithful to the notion that one super-cluster[1] should represent one attack attempt against the network.

We provide a summary of the complete evaluation process we underwent in three steps:

1. We created a set of gold standard clusters by hand for the incidents.org [78] and

[1]See Section 3.3 for an explanation of the term super-clusters.

DARPA [18] datasets.

2. We evaluated our complete system against the incidents.org gold standard clusters to determine close-to-optimal choices for clustering algorithms, algorithm parameters, and other system variables.

3. We applied the choices made at Step 2 to our system and tested it against the DARPA gold standard data. From this test we reported our performance results.

We give more details on the gold standard in Section 4.1.1 and we talk about how we use the gold standard in Section 4.1.2. In the sections on our experiments, Section 4.2 and Section 4.3, we use the evaluation parts of the gold standard, and in the section on overall system performance, Section 4.4, we use the testing part of the gold standard.

Selection of Gold Standard Data

The gold standard is used for testing and evaluation, and was gathered from the incidents.org and 1999 DARPA IDS datasets. We described in Section 3.1.1 why these datasets were chosen. We formed the evaluation part of the gold standard primarily from the incidents.org dataset [78], and we took the testing part of the gold standard from the 1999 DARPA dataset [18].

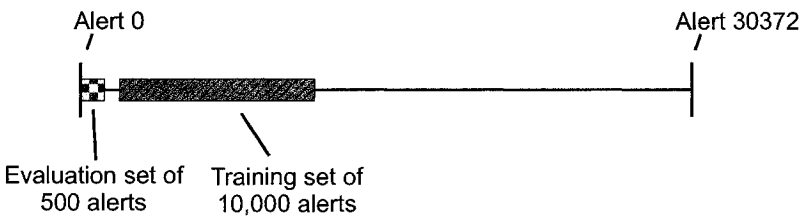

Figure 4.1: Selection of training and gold standard evaluation alerts from the incidents.org data source

In the set of alerts generated by Snort on the incidents.org traffic we selected the first 500 alerts for evaluation. These 500 alerts are the evaluation part of the incidents.org gold standard. We reserved alerts 1000 through 10 999 for training, for a total of 10 000 training alerts. We used the sample of 500 alerts at the beginning because they were

more diverse and interesting to correlate and use as examples. The selection of these datasets is shown visually in Figure 4.1. The 500 alert evaluation set is organized into clusters, as we discuss in Section 4.1.1.

We recognize that to produce reliable test results a proper experiment would be to train on the first 10 000 alerts, then test on the remaining alerts, since alerts appear earlier in the Snort output if they were generated from earlier traffic. Testing on alerts from before the training period could invalid a test since it is not a realistic configuration for a real-world deployment — that is, in the real world the clusterer will always be trained on previously seen data then deployed for future data — but we use this configuration primarily for evaluation. We report the overall performance of the system on the incidents.org evaluation dataset in Section 4.4 for interest's sake, but we rely on the DARPA dataset to provide scientifically sound empirical results.

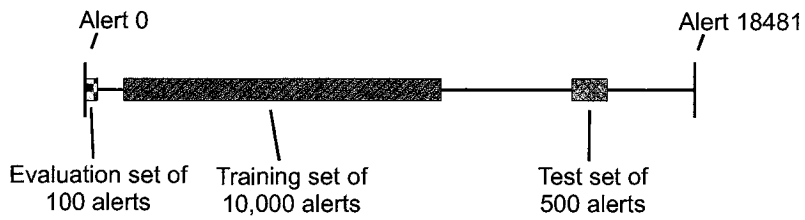

Figure 4.2: Selection of training and gold standard evaluation and testing alerts from the DARPA data source

For the set of alerts generated by Snort on the 1999 DARPA dataset traffic we selected the first 100 alerts for evaluation of the first stage of correlation. These alerts were used in experiments in Section 4.2.2 and Section B.1 before we designated a different part of the 1999 DARPA IDS dataset for the overall performance testing in Section 4.4. We reserved the alerts 1000 through 10 999 for training and we reserved alerts 15 000 through 15 499 for testing the complete system. See Figure 4.2 for a visual description of where the training, evaluation, and testing sets were selected from in the DARPA dataset.

4.1.1 Gold Standard for Performance Evaluation

We used the evaluation part of the gold standard to shape the output of our system to what we wanted to see correlated. We have created an set of clusters by hand as an analyst would[2] from a set of five consecutive windows of 100 alerts from the incidents.org dataset, for a total of 500 alerts in the evaluation part of the gold standard.[3] In creating this set of clusters by hand that we have evaluated our system against, we were trying to force our system output to match the gold standard. Dain *et al* [51] created by hand groupings of alerts to which they compare the output of their system to report on performance, similar to our evaluation and testing configuration.

We include a portion of the gold standard as Appendix A. There are 25 super-clusters in the gold standard, but we include only an abbreviated form of the gold standard super-clusters in the appendix.

Format of Gold Standard

We originally created the gold standard for the evaluation of our complete system based on the hand clusterings we had created for the first correlation stage for each of the five windows of 100 alerts. Since each window of 100 alerts had their own by-hand clusterings, the gold standard represents each cluster in the windows of 100 alerts as a single number. For instance, super-cluster 1 of the gold standard has six members, or sub-clusters: 1.1, 1.2, 1.3, 2.5, 4.12, and 4.13. In this notation, 1.2 stands for cluster 2 of the first window of 100 alerts, 4.12 stands for cluster 12 of 100-alert window 4, etc. This was a convenient way to encode the relationships between clusters within super-clusters.

[2]The gold standard dataset is a set of clusters created by hand by us that represents how we would like our end system correlate the set of alerts. The dataset may contain some errors, just like the output of an analyst would. The dataset represents the correlations we would like our end system to produce.

[3]We also use 100 alerts from the 1999 DARPA dataset for evaluation, as we discussed in Section 4.1. These 100 alerts are not part of the testing part of the gold standard.

Algorithm and Parameter Evaluation

As mentioned, we use the evaluation part of the gold standard to evaluate the performance of our system. In our experiments we compare the performance of multiple variations of our system to choose the system which performs best. In some experiments we select which clustering algorithms should be used at the first or second stage of our system. In other experiments we tune parameters of these clustering algorithms or modify some other implementation detail of the algorithms, such as which data scaling method to use.

After experimenting with different clustering algorithms and sets of parameters, we hold these parameters and algorithms constant for the overall system performance testing in Section 4.4. For example, we find the CLUSTERBARRIER algorithm parameter b is close to optimal if set at 0.0025 in experiments in Section 4.2.3, so we use this value of b in testing. Different values of the parameter may be optimal for different sets of data, but we find that this value is appropriate for the data we considered in our experiments.[4]

If this alert correlation system were deployed in the real world, it would not be possible to know if this value of the parameter b is optimal before correlating alerts from the IDS. Therefore we hold all algorithms and parameters constant for our overall system performance testing. Holding all system parameters constant allowed the system to be tested accurately, since this configuration models the real world.

4.1.2 Results Comparison against Gold Standard

There were multiple choices for the method of evaluation of our performance in the two-stage correlation system. In our correlation system we had the option of testing the two stages separately by scoring how well the second stage of our system does given the first stage. We could have matched clusters produced by our first stage to their gold standard

[4]Note that in our experiments we limit the number of parameters we optimize for in any given experiment. We chose the parameters to optimize based on preliminary tests of which parameters affected the results the most. We acknowledge that a full optimization effort would be required to deploy this system but we consider our system a prototype in which optimization of parameters was not the most important result. We optimize the parameters to the extent that we do to show the capabilities of our system but we admit that better performance could be attained with a greater optimization effort.

clusters, then evaluated how well our system did with respect to the gold standard super-clusters. We chose not to use this system of error-counting, though, because it would have been fraught with problems. There is no perfect match between our gold standard clusters and the clusters produced by the first stage of our system. Inevitably, smaller clusters produced by the first stage of our system would not have been counted, so we wouldn't have had a very realistic evaluation of our system.

We decided to evaluate the performance of our complete system by translating the super-clusters into the set of alerts represented by the sub-clusters in our super-clusters. For instance, using the example given previously, if super-cluster 1 has sub-clusters 1.1, 1.2, 1.3, 2.5, 4.12, and 4.13, and sub-cluster 1.1 has alerts 1, 2, 3, 4, 5, 6, 8, and 10, sub-cluster 1.2 has alert 9, sub-cluster 2.5 has alert 146, etc., then we translate super-cluster 1 into the set of alerts 1, 2, 3, 4, 5, 6, 8, 10, 9, 7, 146, 303, and 324. Once this translation is done, we can compare all of the alerts of our gold standard of super-clusters to the actual results by simply counting the errors.

By reporting our results in this way, we get a good sense of the overall performance of our system, since two alerts are counted as successfully being clustered together if and only if they appear in the same, correct super-cluster.

4.1.3 Types of Errors

We counted the number of errors we made by counting the number of alerts which were correctly clustered versus the number of alerts which were incorrectly clustered for a sample of data.[5] There are other ways of counting errors — such as counting the number of completely correct clusters — but we felt our method was the most accurate gauge of our performance because it ensures that each alert is weighted equally in the performance evaluation. From the perspective of the IDS analyst, large clusters of alerts are likely to warrant more time for analysis than small clusters, so an error counting system that weights each cluster equally is inappropriate. Counting whether each alert is clustered correctly effectively weights larger clusters more.

[5]We use the gold standard from Section 4.1.1 to determine which alerts are clustered incorrectly.

We differentiate between two types of errors in our analysis because they are of different levels of importance, and because they have different causes. We call the first type of error *separation error*, which occurs when a distinct group of alerts found by the candidate algorithm should belong to a larger group of related alerts, but is not clustered with the larger group. For instance, if we have a cluster A of 10 nmap scan alerts and a separate cluster B of 3 nmap alerts that is clearly related to cluster A, then we count 3 separation errors because the 3 alerts in B should have been grouped with the 10 alerts in A. We consider this type of error less damaging[6] because, if this error occurs, the intrusion analyst must analyze more clusters, but he will not miss important alerts hidden in a larger group.

The second type of error we find is *clustering error*, which occurs when two unrelated alerts are clustered together. For instance, if we have a mixed cluster of 10 nmap scans and 5 Squid[7] scans where the alerts of the two types in the cluster are obviously not related, we count 5 clustering errors. This type of error indicates the problem of the system finding correlation where there is none. We consider clustering error more serious than separation error because it may cause an intrusion detection analyst to miss important information, if the analyst is relying on the results of the alert correlation system. Any alert correlation system should only make the intrusion detection analyst's job easier, but clustering errors could make the analyst's job harder by hiding important alerts. More dangerously, an attacker could possibly craft an attack to avoid detection by somehow forcing important alerts to not be seen by the IDS analysts.

4.2 First Correlation Stage Experiments

Here we present the results of experiments we have performed relating to the first stage of correlation in our system. In Section 4.2.1 and Section 4.2.2 we try to find the best algorithms and parameters for the first correlation stage of our system. In Section 4.2.3 we experiment with data different scaling methods and try to determine which one is

[6]We mean "damage" here to be theft of information, monetary loss through time consumed repairing systems or discovering the extent of the intrusion, etc.

[7]Squid is a popular open-source web proxy available at `http://www.squid-cache.org/`.

best.

Note that there are other experiments relating to the first stage of correlation of our system in Appendix B.

4.2.1 Clustering Algorithm Selection

In this experiment we examine different clustering algorithms for possible use in the first stage of correlation of our system. We examine three algorithms: (i) the autoassociator, with the CLUSTERBARRIER algorithm to form discrete clusters, as described in Section 3.4.1, (ii) self-organizing maps, from Section 2.4.3, and (iii) the EM algorithm, described in Section 2.4.3. We evaluate each of the algorithms with different sets of parameters to understand the performance of each clustering algorithm. We discuss the performance results of each algorithm.

The first stage of correlation requires a clustering algorithm to determine the steps of attacks in a set of alerts. As presented in Section 4.2, the first stage clustering algorithm is used to cluster windows of 100 encoded alerts. In this experiment, the three clustering algorithms cluster the five consecutive windows from the incidents.org evaluation gold standard. The results for each algorithm are compared against the gold standard clusters to determine performance.

In this experiment the autoassociator is trained with only 1017 unlabelled alerts from the set of 10 000 incidents.org training data of Section 4.1. This smaller set of 1017 training data was used because there were difficulties in training SOMs on the 10 000 alert set, and it was important that the autoassociator and SOMs were trained on the same data so that they can be compared in this experiment.[8] The set of 1017 training alerts contains roughly the first 10% of each type of alert from the 10 000 alert set, with the number of alerts of each type rounded up if the 10% calculation did not yield a whole number.

As implied, the self-organizing maps are trained on the same set of 1017 training data used

[8]We had problems with the built-in Matlab implementation of SOMs [68] on the 10 000 training data. Other implementations, such as the SOM_PAK implementation by Kohonen *et al* [74], were experimented with as well, but they were not satisfactory.

by the autoassociator. The EM algorithm was not trained at all, because this clustering algorithm is not designed to be trained with unlabelled data. (See Section 3.4.3 for details.)

We generated results for the autoassociator with 8, 16, 32 and 64 hidden units at epochs in the range $[25, 5000]$ at 25 epoch intervals. The results for the autoassociator are presented in Figure 4.3. We generated results for the EM algorithm with the number of clusters set at $1, 5, 10, \ldots, 60$, etc. The results for the EM algorithm are shown in Figure 4.4. We generated results for the self-organizing map algorithm with lattice dimensions of 4×6, 5×7, and 6×8 at epochs in the range $[25, 5000]$ at 25 epoch intervals. The results for the SOM algorithm are displayed in Figure 4.5.

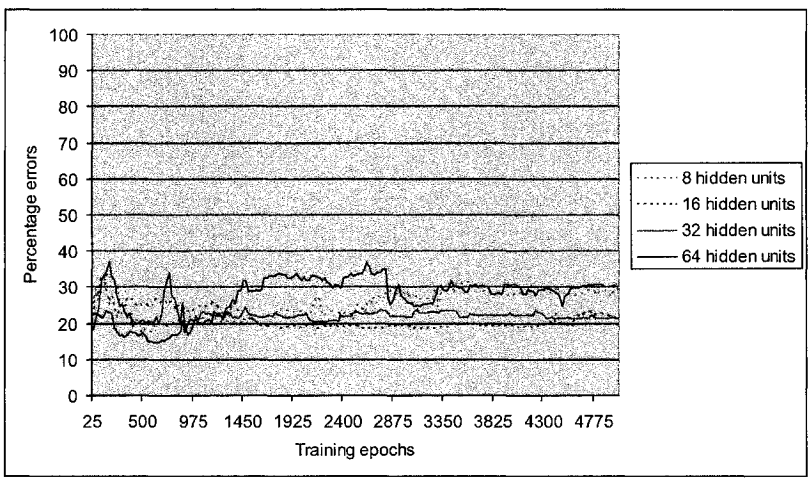

Figure 4.3: First stage performance with autoassociator

In Figure 4.5 it is clear that the autoassociator attains the best performance if constructed with 64 hidden units and if trained for between 500 and 1000 epochs. The results with 64 hidden units are the worst at higher epoch values and the results with 16 hidden units are the best and most consistent at higher epoch values, but we are searching for the best point of performance in this experiment.

As you can see for the performance graph of the EM algorithm in Figure 4.4, if the number of clusters chosen is 10 or more, the performance results are consistently in the range 30% to 40% errors. The best performance for the EM algorithm are at 30 clusters, but the

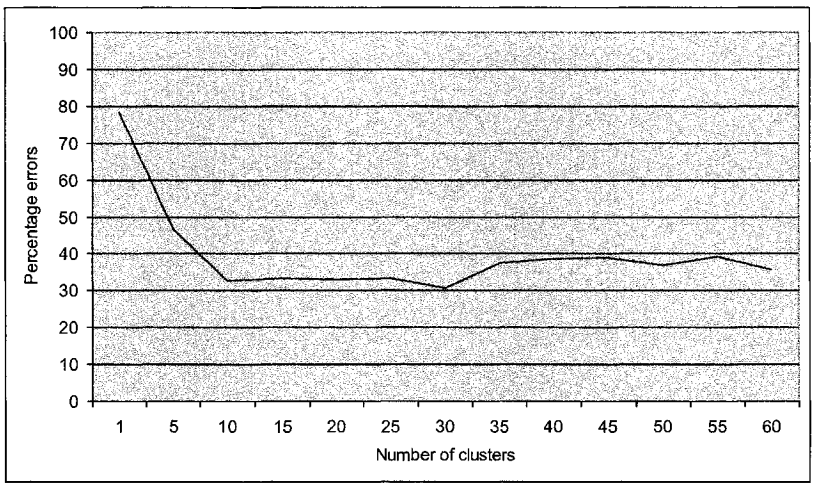

Figure 4.4: First stage performance with the EM algorithm

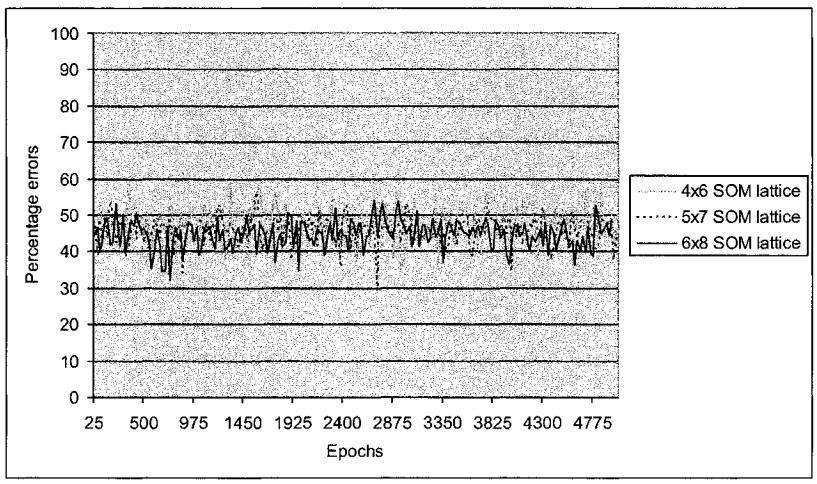

Figure 4.5: First stage performance with self-organizing maps

results with this algorithm are consistently not as strong as those of the autoassociator in Figure 4.3.

In Figure 4.5 we present the results of the self-organizing maps clustering the IDS alert data. The results for the SOM algorithm are noticeably worse than the results with the autoassociator and the EM algorithm. The results for all three lattice configurations vary significantly with no discernable range of best performance. The mean performance for the 4×6 for all points in Figure 4.5 is 48% errors, the mean percentage errors for the 5×7 points is 45.3%, and the mean percentage errors for the 6×8 points is 44.8%. From this we see that the performance of SOMs is universally poor, especially if compared with the autoassociator, which does not generate an error percentage greater than 40% in this evaluation.

From this clustering algorithm evaluation, we can conclude that the autoassociator performs better than the EM algorithm and SOMs for this dataset against the incidents.org gold standard evaluation dataset. We also found that the autoassociator performs best if constructed with 64 hidden units and if trained for between 500 and 1000 epochs. We recommend training the autoassociator for 500 epochs as a method of overfitting avoidance (see Section 2.4.2), since training for fewer epochs is less likely to produce an overfitted neural network.

Autoassociator results with 10,000 training examples

As discussed, the autoassociator was trained with 1017 unlabelled training data for the results previously presented in this section. We include the results of the autoassociator trained with the full 10,000 item training set here as well, since we use this training set for other experiments and our performance results in Section 4.4. We present the results with the 10,000 item training set in Figure 4.6.

It is clear when visually comparing the results with 10,000 training data, in Figure 4.6, to the results with 1017 training data, in Figure 4.3, that the size of the training set has very little effect on the results with this particular evaluation set. As you can see, the lines representing the results for 64 hidden units in the two graphs are very similar. The

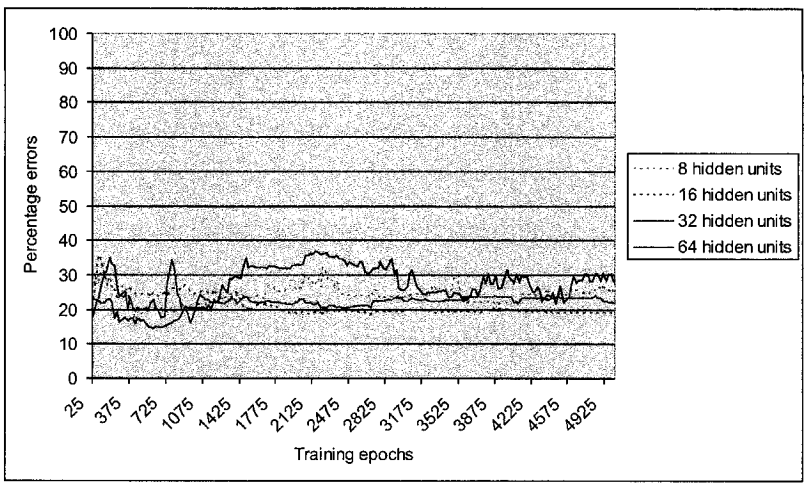

Figure 4.6: First stage performance with autoassociator trained on 10,000 unlabelled data

analysis that applies to Figure 4.3 also applies to Figure 4.6.

4.2.2 Autoassociator Parameter Exploration

In this experiment we want to discover the best parameters for the autoassociator at the first correlation stage of our system, described in Section 3.2. We examine two autoassociator parameters, hidden units and epochs, in these experiments to determine the best performance of our system. We also explore which of three candidate clustering algorithms performs best in clustering the autoassociator output.

Recall that clustering data with a trained autoassociator requires two steps: simulating the data on the neural network then clustering the output of the network. We experiment with different ways of clustering the output of the autoassociator in this experiment. To better understand the configuration of this experiment, it may be useful to refer to Figure 3.3. We are experimenting with the clustering algorithm in this figure by evaluating different clustering algorithms rather than just the single-link CLUSTERBARRIER algorithm. We are also experimenting with the autoassociator by evaluating the neural network with different numbers of hidden units and epochs.

We compare the performance in terms of errors produced by various combinations of unsupervised learning algorithms and parameters on two datasets. Namely, we varied the autoassociator's neural network hidden units and epochs parameters, and we fed the 40 autoassociator outputs into three clustering algorithms: the EM algorithm, self-organizing maps (SOM), and the single-link clustering algorithm that we presented in Section 3.4.2.

To compare the performance of the various modifications to our system, we have used one window of 100 alerts from each of the 1999 DARPA and incidents.org datasets. The 100 alerts from the 1999 DARPA dataset are the DARPA gold standard evaluation data of Section 4.1, and the 100 alerts from the incidents.org dataset are the first 100 alerts of the incidents.org gold standard evaluation data.

As discussed in Section 2.4.2 and Section 3.4.1, the hidden units and epochs of the autoassociator are important parameters in training. In this experiment we considered the following four candidate hidden units values: 8, 16, 32, and 64. We examined three candidate epoch values for the autoassociator: 500, 2500, and 5000.

Since the self-organizing map algorithm has parameters to be chosen, we include the values we have chosen for them. The number of training epochs is fixed at 1000 and the dimensions of the map lattice are 4×6. According to Kohonen *et al* [74], these parameter choices are appropriate, and this was confirmed in other experiments we ran.

In this experiment we compare the results of clustering the incidents.org evaluation dataset of 100 alerts using a trained autoassociator and three different clustering algorithms: the single-link clustering algorithm from Section 3.4.2, the EM algorithm, and self-organizing maps. In the following tables, we list the total percentage errors (TE), the percentage clustering errors (CE), and the percentage separation errors (SE). (Note that $TE = CE + SE$, since clustering errors and separation errors are the only two types of errors that occur in our system.)

We see from Table 4.1, Table 4.2, and Table 4.3 that the lowest percentage of total errors we achieved is 28%, achieved by SOMs clustering the output of the autoassociator with 32 hidden units trained for 2500 and 5000 epochs. We see that this system, if trained for 2500 epochs, has fewer clustering errors than the system trained for 5000 epochs. Since

Table 4.1: Results of the autoassociator and the single-link algorithm on incidents.org alerts

	Epochs=500			Epochs=2500			Epochs=5000		
Hidden Units	TE	CE	SE	TE	CE	SE	TE	CE	SE
8	29	8	21	37	3	34	37	3	34
16	47	8	39	48	8	40	47	6	41
32	42	2	40	43	1	42	38	0	38
64	40	0	40	40	1	39	43	5	38

Table 4.2: Results of the autoassociator and the EM algorithm on incidents.org alerts

	Epochs=500			Epochs=2500			Epochs=5000		
Hidden Units	TE	CE	SE	TE	CE	SE	TE	CE	SE
8	32	4	28	39	3	36	43	2	41
16	44	0	44	44	0	44	44	0	44
32	39	2	37	40	0	40	46	2	44
64	39	2	37	43	1	42	38	1	37

Table 4.3: Results of the autoassociator and SOMs on incidents.org alerts

	Epochs=500			Epochs=2500			Epochs=5000		
Hidden Units	TE	CE	SE	TE	CE	SE	TE	CE	SE
8	34	6	28	33	4	29	31	3	28
16	33	4	29	33	4	29	34	4	30
32	29	4	25	28	2	26	28	5	23
64	35	4	31	30	4	26	30	4	26

we regard clustering errors as more damaging than separation errors, we can conclude that the best system performance in this experiment was attained by the self-organizing maps with an autoassociator trained for 2500 epochs.

To analyze the effectiveness of the three clustering algorithms, it might be useful to consider the averages of TE, CE, and SE from Table 4.1, Table 4.2, and Table 4.3, to show the general trends of the algorithms. We show these averages in Table 4.4. The values in *italics* are the variance measures of the total errors.

Table 4.4: Clustering algorithm average errors for incidents.org alerts

	Average Errors		
Clustering Algorithm	TE	CE	SE
Single-link	40.9 (*28.8*)	3.8	37.2
EM	40.9 (*14.8*)	1.4	39.5
SOM	31.5 (*6.1*)	4	27.5

We see from Table 4.4 that SOMs clearly have the lowest overall average number of errors. Another interesting point of this table is that the EM algorithm produces a lower average number of clustering errors than the other two clustering algorithms.

Next, we apply the same selection of algorithms and parameters to test which algorithm performs best with 100 alerts from the 1999 DARPA part of the gold standard evaluation dataset. In Table 4.5, Table 4.6, and Table 4.7 we see that all of the clustering algorithms perform more poorly than on the incidents.org gold standard evaluation dataset. The reason for the poorer performance in this experiment is that correlations in the 1999 DARPA gold standard evaluation dataset differs from those in the incidents.org dataset.

Table 4.5: Results of autoassociator and the single-link algorithm on DARPA alerts

	Epochs=500			Epochs=2500			Epochs=5000		
Hidden Units	TE	CE	SE	TE	CE	SE	TE	CE	SE
8	52	6	46	52	0	52	52	4	48
16	52	2	50	50	5	45	48	1	47
32	51	4	47	50	2	48	51	0	51
64	48	0	48	50	0	50	50	0	50

While we may not be able to achieve precisely the gold standard for this dataset, the clusters produced by the systems are still valuable to an IDS analyst. For instance, we

Table 4.6: Results of autoassociator and the EM algorithm on DARPA alerts

	Epochs=500			Epochs=2500			Epochs=5000		
Hidden Units	TE	CE	SE	TE	CE	SE	TE	CE	SE
8	44	3	41	61	0	61	62	3	59
16	63	4	59	43	3	40	43	3	40
32	44	8	36	51	4	47	60	3	57
64	65	3	62	45	3	42	64	3	61

Table 4.7: Results of autoassociator and SOMs on DARPA alerts

	Epochs=500			Epochs=2500			Epochs=5000		
Hidden Units	TE	CE	SE	TE	CE	SE	TE	CE	SE
8	62	4	58	61	4	57	57	7	50
16	61	3	58	61	2	59	59	3	56
32	58	6	52	60	9	51	59	2	57
64	60	9	51	58	6	52	58	5	53

see that the percentage clustering errors are low, implying that the clustering algorithms found differences within clusters of the gold standard.

We note that our best performance lies with the EM algorithm for this dataset, with 16 hidden units in the autoassociator. The system performs equally well when the autoassociator is trained for both 2500 and 5000 epochs.

Although the EM algorithm produces the single best system for the 1999 DARPA gold standard evaluation dataset, the results produced by this algorithm vary more significantly than the results produced by SOMs and the single-link clustering algorithm, indicating that the results produced by the EM algorithm may vary more in general, indicating a lack of stability. We know that the EM algorithm was responsible for the best performing system, but we see in Table 4.8 that the single-link clustering algorithm outperformed the EM algorithm on average.

From Table 4.8 we see that the single-link algorithm has both the lowest average total errors as well as the lower average clustering errors. Another interesting fact presented in this table is that SOMs performed the poorest on this dataset. They had both the highest average total error and the highest average clustering error. We find this interesting, because they performed the best on the incidents.org dataset.

Table 4.8: Clustering algorithm average errors for DARPA alerts

	Average Errors		
Clustering Algorithm	TE	CE	SE
Single-link	50.5 (*2.1*)	2	48.5
EM	53.8 (*89.3*)	3.3	50.4
SOM	59.5 (*2.5*)	5	54.5

From the SOM results in Table 4.3 and Table 4.7 we see that the percentages vary less than the other results. Table 4.4 and Table 4.8 report the variance in the total error statistic for the three algorithms on the two datasets. From these tables we can hypothesize that SOMs form more stable clusters, which are less likely to be changed by the intermediate step of using the autoassociator. This could be considered either a positive or negative trait of the algorithm. Stability of results is positive because it can imply a predictable end system, but it can be considered negative if the algorithm is inflexible and cannot be tuned.

We note in these experiments that the single-link algorithm attains better performance if the autoassociator is constructed with 32 or 64 hidden units. One can see in Table 4.1 and Table 4.5 that with 64 hidden units at 500 epochs, in particular, the single-link algorithm produces few clustering errors. This is a desirable property of a clustering algorithm candidate for this problem.

Another desirable trait of the autoassociator algorithm combined with the single-link ClusterBarrier algorithm is that the number of clusters to output do not need to be specified before the clustering begins. This value is determined automatically with the ClusterBarrier algorithm, influenced by the cluster barrier parameter b.

From the analysis of the experiments in this section, we cannot conclude that one clustering algorithm is clearly more effective for use with the autoassociator, but we favour the single-link ClusterBarrier algorithm because of automatic determination of the number of clusters to form.

4.2.3 Data Scaling Algorithms

In this experiment we evaluated different scaling methods, particularly those discussed in Section 3.1.3. In artificial neural networks it is often important to scale data so that computational errors do not occur in neural network training and so that features are weighted equally. In our experiments we evaluated different scaling methods with respect to system performance.

The experiments we describe here are intended to explore two problems. The first problem we explored is the importance of scaling the data. We show the importance of data scaling by showing how our system performs with unscaled data. We compared the results of using unscaled data to the results when using scaled data. The second problem we explored is which scaling method performs best. The scaling algorithms we considered make the overall system perform differently and we find that the ClusterBarrier algorithm parameter b needs to be adjusted for each different algorithm. We vary this parameter for our experiments to give an idea of the potential for each method.

For the results of this section we used the evaluation clusterings from the five windows of 100 alerts from the incidents.org dataset from the gold standard, in Section 4.1.1. We determined the performance of our system on each of these five evaluation sets and we averaged the errors produced with the five datasets to gain a better view of how the system performs. The average number of errors is the percentage of the total possible number of errors. The system with the lowest percentage of errors is considered the best.

We expected that we'd have to adjust the cluster barrier parameter b for this experiment by necessity because this parameter is directly dependent on how the data is encoded and scaled. We explore this parameter graphically by showing the percentage of total errors, clustering errors and separations errors plotted against the value for b for the system in question.

Figure 4.7 shows how the first stage of our system performs when scaling features to the interval $[0, 1]$ using feature ranges determined from the training set. As you can see from this graph, the best performance is attained with the value 0.0017 of b, which is fairly close to the value used in our system, 0.0025.

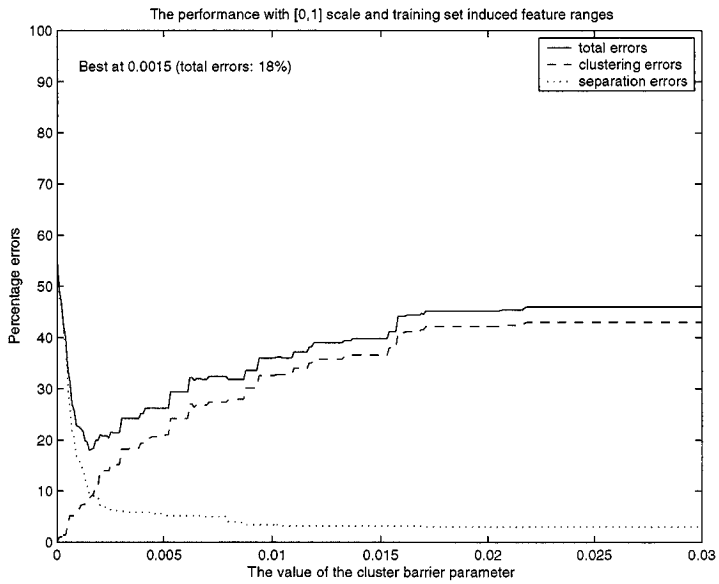

Figure 4.7: First stage performance using a [0,1] scale with training set-determined ranges

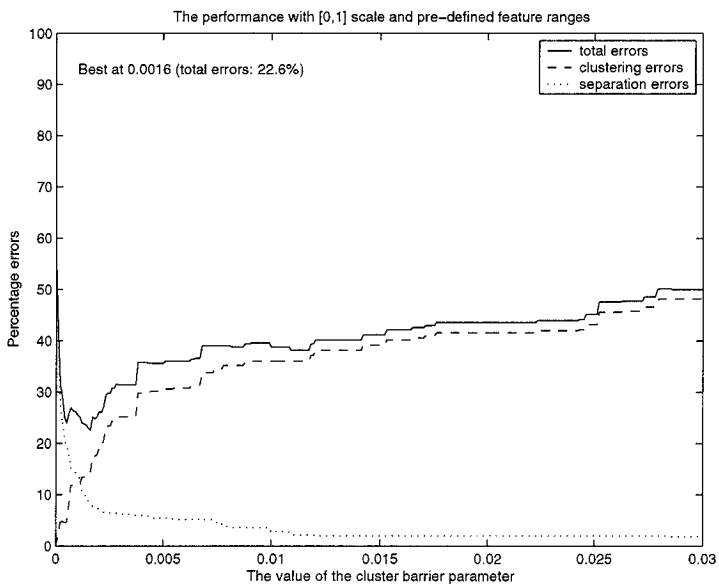

Figure 4.8: First stage performance using a [0,1] scale and predefined ranges

We present another variation of our first stage scaling to the interval $[0, 1]$ in Figure 4.8. For this graph we used the predefined feature ranges we discussed in Section 3.1.3. In this figure we see that this system performs noticeably worse than the other system that scales linearly to the interval $[0, 1]$.

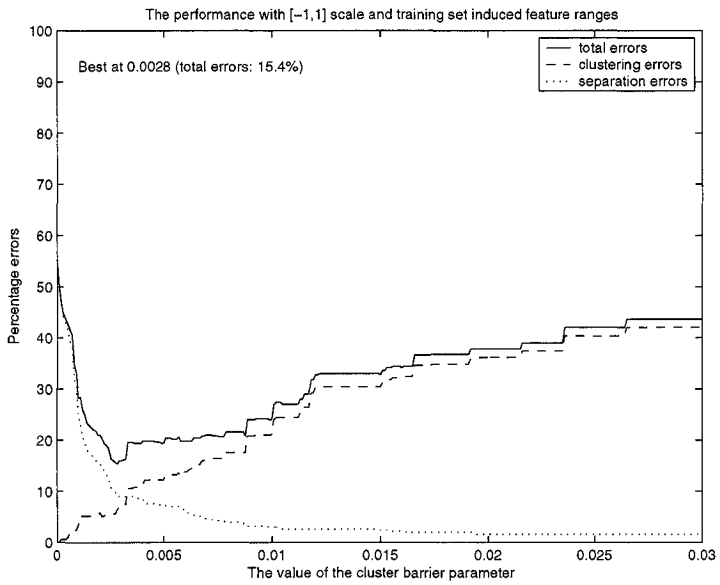

Figure 4.9: First stage performance using a [-1,1] scale with training set-determined ranges

Figure 4.9 shows the performance for the system scaling to the interval $[-1, 1]$ using feature ranges determined from the training set. Figure 4.10 shows the performance for this same interval but with the predefined feature ranges. Figure 4.10 showed the lowest error rate of any of the systems we examined, and Figure 4.9 showed error rates so close to this that the difference isn't statistically significant. The two graphs show essentially the same performance, and they show this with almost the same value for the cluster barrier parameter, 0.0028 and 0.0027.

Figure 4.11 shows how well our system performed with the Gaussian scaling method from Section 3.1.3. As you can see from this graph, the best results are not as good as the other methods, but it seems that the performance was less affected by the cluster barrier parameter b, because larger values for this parameter didn't produce significantly worse

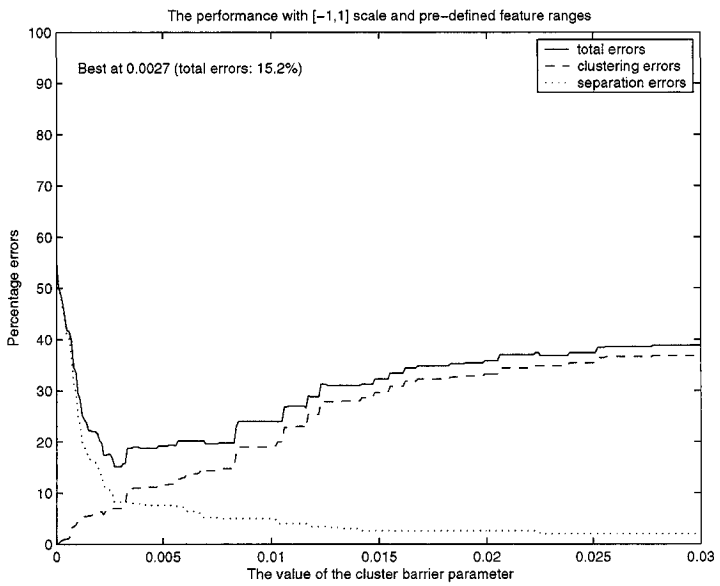

Figure 4.10: First stage performance using a [-1,1] scale and predefined ranges

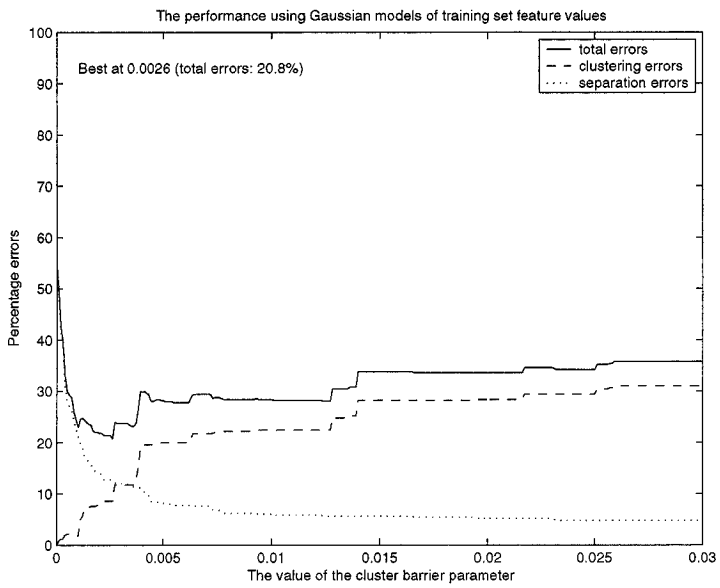

Figure 4.11: First stage performance using training set-derived Gaussian models for features

results, unlike in the other systems. We found this interesting, but we do not choose this system of scaling because we plan to fix the cluster barrier parameter at a lower value anyway.

Lastly we tried running our system on unscaled datasets. The results were so poor[9] that we do not present them in graphical form here. The results hardly changed at all when different values for the cluster barrier parameter b were evaluated.

We conclude that selection of which scaling method to use is important. We also conclude that scaling linearly to the interval $[-1, 1]$, using either of the methods for which we presented results, with a cluster barrier parameter value of about 0.0025 produces the best results.

4.3 Second Correlation Stage Experiments

The experiments in Section 4.3.1 and in the performance testing Section 4.4 are the only experiments that consider the entire system. Since the experiments in Section 4.2 consider only the first stage of correlation, this section of experiments takes the results of the first stage experiments as a given.

4.3.1 Algorithm and Parameter Selection

For this experiment we explored the clustering algorithms available to us and their respective parameters to find a system that produced the best possible clusters for the second correlation stage. We defined the performance of the systems we explored as the percentage of errors the system produced with respect to the incidents.org gold standard evaluation dataset of 500 alerts from Section 4.1.1.

In our experiments with the first correlation stage of our system, we found that if we used a base window of 100 alerts, we would produce about 25 clusters with our system. Obviously this number of clusters would vary with the dataset we're using (and thus

[9]The best performance attained for the system while using unscaled data had an error rate of 55%.

with the number of innate clusters in the data), but we found this to be a good estimate in general. For the second correlation stage of our system, where each cluster produced by the first stage becomes its own data item, finding correlation in a set of only 25 data items wouldn't give us meaningful results. So we decided to run our first correlation stage, with the parameters we discovered in experiments in Section 4.2, on the five consecutive windows of 100 alerts that were used to create our incidents.org gold standard in Section 4.1.1. Once the first correlation stage had produced clusters for each of these five windows, we ran our new feature construction methods on each of the new clusters then combined all of the data produced into one dataset.

Running the first correlation stage on these five windows produced 83 clusters for our new dataset. These 83 clusters were converted into 83 data items with our second correlation stage feature construction methods. Since the purpose of this experiment was to explore feature construction for this dataset, we held the number of data items in the set static, and experimented by analyzing the output when we selected particular features that we've constructed.

For this experiment we evaluated two sets of features for the second correlation stage of our system. We tested our system with all of the features we discussed in Section 3.3.1 and we also tested our system with only the features adopted from Dain and Cunningham system [51].[10] For the latter test we used the following subset of features: `ipSrcAddrCommonPart`, `ipDestAddrCommonPart`, `ipSrcAddrCommonBits`, `avgTimeSig`, `varTimeSig`, and `avgReconsErr`. For the former test we included all of the Dain features as well as some others, to form the feature set: `ipSrcAddrCommonPart`, `ipDestAddrCommonPart`, `ipSrcAddrCommonBits`, `avgTimeSig`, `varTimeSig`, `avgReconsErr`, `modePortSrc`, `modePortDest`, and `avgSeqNumDiff`.

We explored which clustering algorithms perform best for our new dataset. To this end, we experimented with all of: the autoassociator [4], the EM algorithm [72], and self-organizing maps [73]. We were guided by some of the results from the experiments of Section 4.2.2, since these algorithms were considered in those experiments as well. For instance we learned that the autoassociator, using the single-link, one-dimensional ClusterBarrier clustering algorithm to form the actual clusters, seems to work well

[10]We call the subset of features from Dain and Cunningham the "Dain features" from here on.

with 64 hidden units. Also, we used a 4×6 lattice for the SOMs, since this worked well previously.

It is not possible to use training data for this problem because our first stage training data is unlabelled, and thus has no discrete clusters from which would could construct our new features. So we did not use training data as such for any of the experiments here. We did however train the autoassociator and self-organizing maps on the data that we were about to cluster, as we've experimented with previously. We held the number of epochs for training the autoassociator at 500, which we found to work well in Section 4.2.2.

We varied one parameter for each of the three algorithms we present here. For the autoassociator and single-link CLUSTERBARRIER algorithm, we were interested in observing the effect of varying the cluster barrier parameter b (previously held constant at 0.0025) because in our initial experimentation with the second correlaton stage we found that this parameter had the most control over performance. We tested this parameter for values between 0.0 and 0.03, at increments of 0.0001.

With self-organizing maps, we varied the number of epochs for which we trained. We reported the performance for all values between 0 epochs and 5000 epochs, at 50 epoch intervals.

For the EM algorithm, the number of clusters to form was the parameter with which we experimented. We varied this parameter between the values of 5 and 60, testing at increments of 5.

We present the results of the system using only the Dain features. The performance of the autoassociator is listed in Figure 4.12, the performance of the self-organizing maps is in Figure 4.13, and the performance of the EM algorithm is in Figure 4.14.

As you can see from Figure 4.12, varying the cluster barrier parameter produces more predictable results than the results in either of Figure 4.13 or Figure 4.14. That said, the best performance attained by the autoassociator under these conditions was 57.4% errors, which is more than the average percentage errors for the entire set of self-organizing maps tests. The autoassociator produced noticeably worse results than the EM algorithm in this experiment as well. SOMs and the EM algorithm had similar performance results for

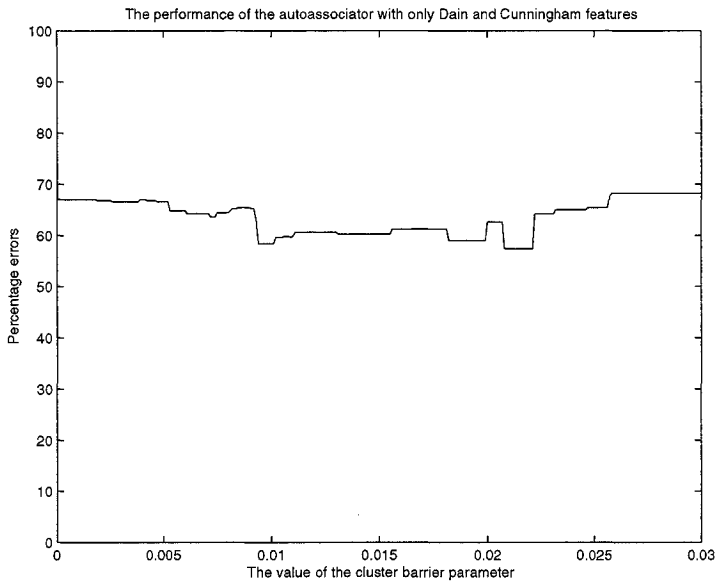

Figure 4.12: Performance with Autoassociator varying cluster barrier for Dain features

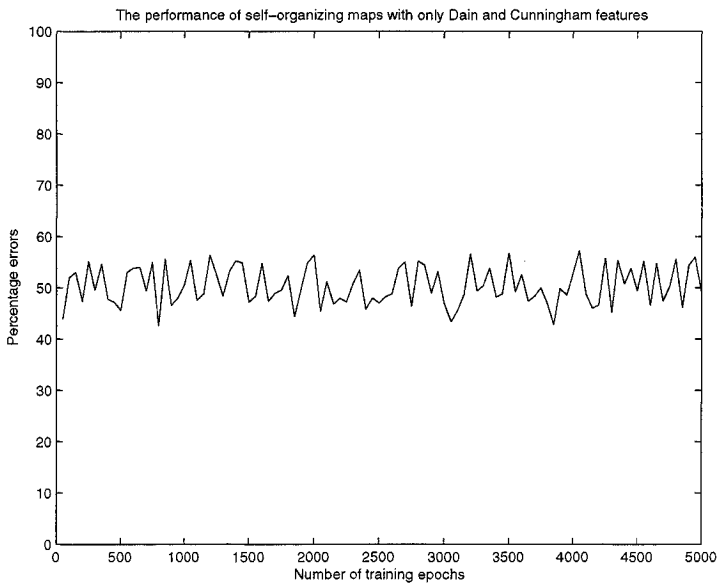

Figure 4.13: Performance with SOMs varying epochs for Dain features

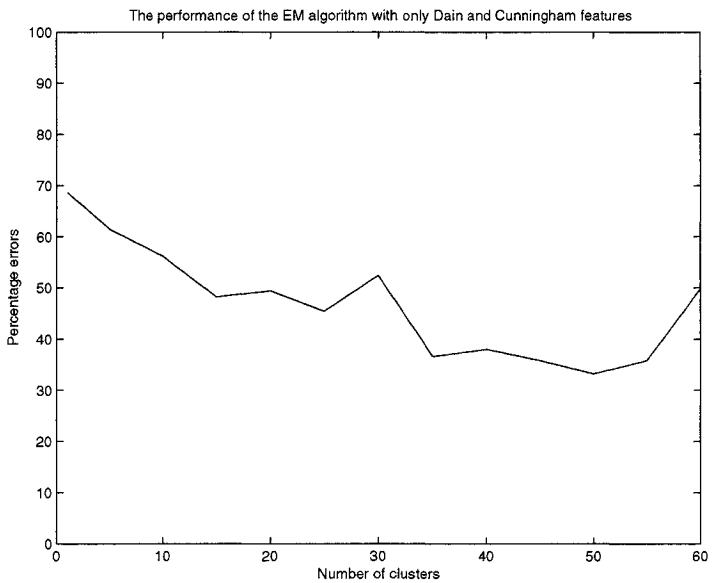

Figure 4.14: Performance with EM algorithm varying number of clusters for Dain features

this experiment — though the EM algorithm attained a slightly lower average number of errors, and a lower best error rate.

We were alarmed by the unpredictability of the results of the self-organizing maps. We expected a smoother curve, rather than one that varies so much between neighbouring epoch values. A curve like that seen in Figure 4.13 indicates a lack of stability in the results because it indicates a lack of predictability. As a consequence of this unpredictability, combined with the fact that the EM algorithm attains better overall performance, we can conclude that the EM algorithm exhibits the best performance for this experiment. The best performance of the EM algorithm is seen with the number of clusters set at 50, but we see interesting results with a number of clusters of 35.

Next we present the results of our system on all of the features from Section 3.3.1. The performance of the autoassociator is listed in Figure 4.15, the performance of the self-organizing maps is in Figure 4.16, and the performance of the EM algorithm is in Figure 4.17.

As you can see from the graphs, the same sort of trends occur that we saw in the previous

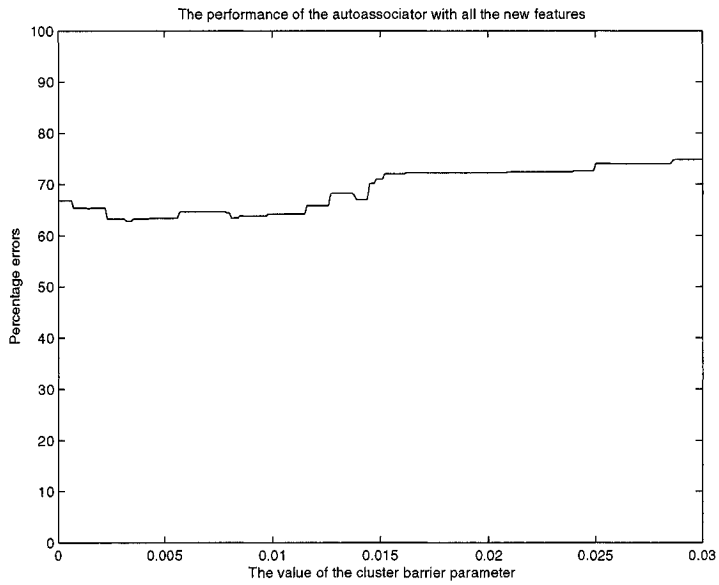

Figure 4.15: Performance with Autoassociator varying cluster barrier for all features

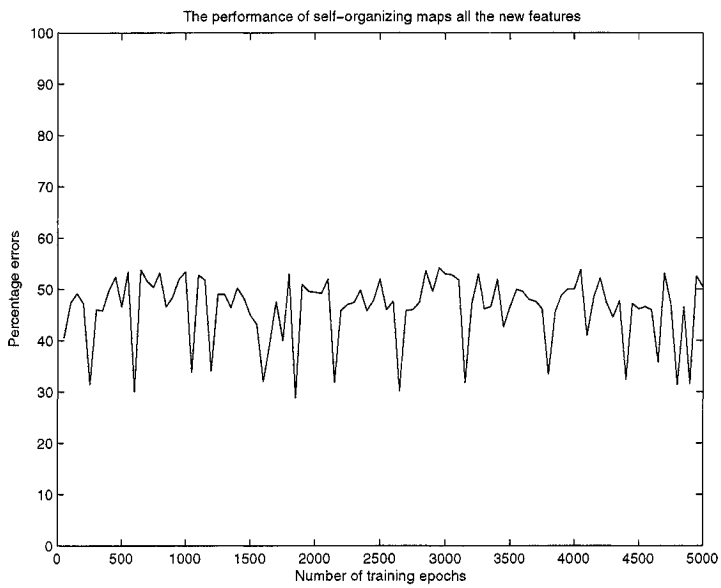

Figure 4.16: Performance with SOMs varying epochs for all features

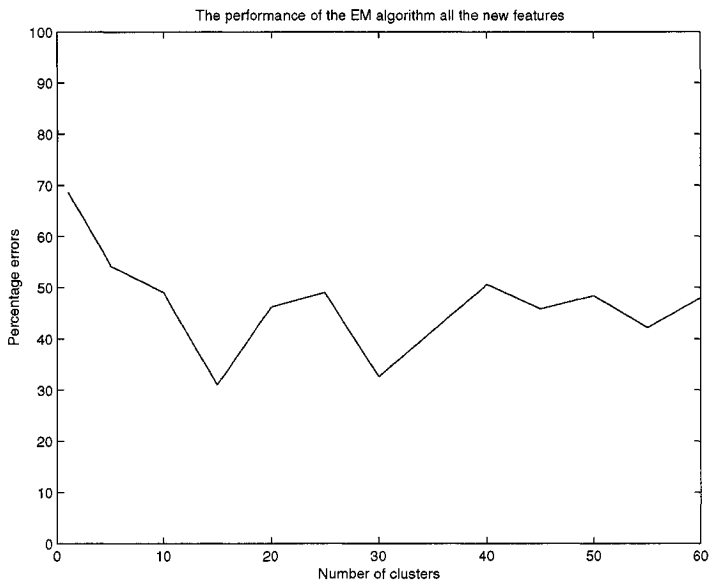

Figure 4.17: Performance with EM algorithm varying number of clusters for all features

set of graphs. When comparing the graph for the autoassociator with the Dain features (Figure 4.12) to the graph for the autoassociator with all the features (Figure 4.15) we see in the former graph that the autoassociator performs better with larger values for the cluster barrier parameter (best value found: 0.021), whereas for the latter graph, the performance peaks at at smaller values for this parameter (best value found: 0.003).

For the pair of graphs representing the performance of self-organizing maps (Figure 4.13 and Figure 4.16) we see that the results were both very volatile. Just as with the Dain feature set, the all features set produced results that were unpredictable. One important thing to note between these two graphs, though, is that the latter graph clearly shows that self-organizing maps perform better with all the features than with only the Dain features.

For the pair of graphs representing the performance of the EM algorithm (Figure 4.14 and Figure 4.17) we see that the overall shapes of the graphs are very similar, and, indeed, the performance shown in the two graphs is similar as well. The latter graph shows that the best performance happens at 15 clusters and 30 clusters. The most interesting results are

seen at the 30 cluster mark. The results at this point seem to balance clustering errors versus separation errors well.

We delved a bit deeper into the results to perform more thorough analysis. Some of the best results we've seen are using the EM algorithm with all features selected and with the number of clusters fixed at 30. With the number of clusters set to 30, the system produced 58 clustering errors and 105 separation errors, for a total of 163 errors, which is a percentage error of 32.6%.

A significant portion of the errors occur because of the problems of two super-clusters. One super-cluster is composed of two separate super-clusters from the gold standard. The two super-clusters from the gold standard are homogenously composed of `nmap` alerts. These two gold standard super-clusters are grouped together by the EM algorithm, and this accounts for 55% of the clustering errors. Also, 41% of separation errors occur because a large super-cluster from the gold standard (composed of "BAD-TRAFFIC tcp port 0 traffic" alerts) is split into two large clusters. Another 20% of separation errors occur because another super-cluster of the same type of alerts is split from its gold standard super-cluster.

From this analysis we can conclude that many of the errors that we've encountered in the results are neither critical nor dangerous. We have seen that some of the errors from the first stage of correlation have propagated upwards and caused clustering errors in the larger super-clusters where there would otherwise not be any errors as well.

To conclude we found that the EM algorithm produces the most interesting and stable results. We found the surprising result that the autoassociator is ill-suited to the second correlation stage, and we found that self-organizing maps produce unpredictable results.

We found the result that the autoassociator is ill-suited to clustering second stage data surprising because the autoassociator performed well for the first stage of correlation. We discuss our interest in performing more experiments to determine for which conditions and types of data the autoassociator as a clustering algorithm is well-suited in Section 5.1.

4.4 Overall Performance Results

In this section we want to report results for the performance of our overall system. To present the results fairly we felt it was necessary to compare our system to the results of another system. To this end, we created the NAIVECORRELATOR algorithm to correlate alerts, which we use as a base line for the performance of our system. The NAIVECORRELATOR algorithm groups together similar alerts based on the types of the alerts, the source IP addresses of the alerts, and the times at which the alerts were reported. The NAIVECORRELATOR algorithm is presented in detail in Section 3.4.4.

After creating the simple correlation system, we tested it against the incidents.org alerts from the evaluation part of the gold standard. We found that there were 292 errors for the evaluation dataset of 500 alerts using this simple correlation scheme. Accordingly, the accuracy of the simple correlation system for the dataset is 41.6%. From Section 4.3.1 we saw that our system had 163 errors for the incidents.org gold standard data, which is an accuracy percentage of 67.4%. So, our system generated 44.2% fewer errors than the naive correlation algorithm on the evaluation dataset.[11]

We also tested our system on the DARPA dataset described in Section 3.1.2.[12] We found with this less complex dataset that our system attained an accuracy of 88.2%. In comparison, the NAIVECORRELATOR algorithm attained an accuracy of 79.4%. So, our complex correlation system generated 42.7% fewer errors than the naive correlator. We present the details of the clusters formed for the DARPA dataset in Table 4.9.

From Table 4.9 we see that the majority of alerts in the DARPA test dataset are of the types "Tiny Fragment", "SNMP public request/access", or "Apache Directory Disclosure". These three classes of alerts account for 95% of the alerts in the dataset. For these three classes of alerts, our system has an accuracy of 89.1%. For the remaining 5% of alerts, our system is 72% accurate. Most of the errors seen in this dataset —

[11]We know that the results of our system on the evaluation dataset cannot be seen as a good indication of our performance because the parameters were tuned on this dataset. We include these results here for interest's sake, since the incidents.org has interesting correlations to be found. We consider the results of testing against the 1999 DARPA IDS dataset the true performance results.

[12]The algorithms and parameters found in the experiments in Section 4.2 and Section 4.3 are held constant in this test.

Table 4.9: System-produced clusters versus gold standard clusters

System-Produced Clusters	Cluster Description	Gold Standard Clusters													
		Apache directory disclosure	SNMP access	Tiny Fragments	telnet access 1	telnet login incorrect	telnet access 2	CGI redirect	CGI calendar	Web /doc access	IIS register.asp	CGI count.cgi	Web 403 forbidden	Web search.dll	Error
1	telnet access 1	0	0	0	*1*	0	0	0	0	0	0	0	0	0	100%
2	Apache directory disclosure	**74**	0	0	0	0	0	0	0	0	0	0	0	0	0%
3	telnet access 2	0	0	0	0	0	0	*1*	0	0	0	0	0	0	100%
4	telnet login incorrect	0	0	0	0	**2**	0	0	0	0	0	0	0	0	0%
5	telnet access 3	0	0	0	0	0	*1*	0	0	0	0	0	0	0	100%
6	telnet access 4	0	0	0	0	0	**2**	0	0	0	0	0	0	0	0%
7	SNMP access 1	0	*6*	0	0	0	0	0	0	0	0	0	0	0	100%
8	Web 403 forbidden 1	0	0	0	0	0	0	*1*	0	0	0	0	0	0	100%
9	telnet access 5	0	0	0	*1*	0	0	0	0	0	0	0	0	0	100%
10	Tiny Fragments	0	0	**225**	0	0	0	0	0	0	0	0	0	0	0%
11	SNMP access 2	0	**124**	0	0	0	0	0	0	0	0	0	0	0	0%
12	SNMP access 3	0	*22*	0	0	0	0	0	0	0	0	0	0	0	100%
13	SNMP access 4	0	*24*	0	0	0	0	0	0	0	0	0	0	0	100%
14	Web search.dll	0	0	0	0	0	0	0	0	0	0	0	0	**2**	0%
15	Web /doc access	0	0	0	0	0	0	0	0	**1**	0	0	0	0	0%
16	IIS register.asp	0	0	0	0	0	0	0	0	0	**2**	*1*	0	0	33%
17	Web 403 forbidden 2	0	0	0	0	0	0	0	0	0	0	0	**1**	0	0%
18	CGI calendar	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0%
19	Web 403 forbidden 3	0	0	0	0	0	0	*1*	0	0	0	0	0	0	100%
20	CGI redirect	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0%
21	telnet access 6	0	0	0	**6**	0	0	0	0	0	0	0	0	0	0%
Error (percentage)		0%	30%	0%	25%	0%	33%	75%	0%	0%	0%	100%	0%	0%	10%

approximately 88.1% — occur for the "SNMP public request/access" class of alerts.

Something to note, though, with the NAIVECORRELATOR algorithm is that it only produced separation errors and did not produce any clustering errors. This is significant because if it is true in general, the NAIVECORRELATOR correlation algorithm could be used to reduce the number of alerts before the datasets are processed by our system.

Clearly our system performs better than the simple base line that we've created. It could be argued that the simple correlation system that we have created is too simple to compare to our complex, optimized system. But to that we note that many of the correlation methods used by IDSs are as simple as the one we presented here. Therefore we believe the comparison is fair.

Chapter 5

Conclusions

Our alert correlation system finds relationships between alerts to assist the IDS analyst in determining the motivation of attackers. The output of IDSs is considered low-level, since a single attack can be represented by many alerts. Our alert correlation system helps to remedy this problem by grouping together alerts that are related. Our system is implemented in two stages. The first stage of correlation groups together alerts such that the groups form steps of attacks. The second stage of correlation groups together the clusters formed at the first stage such that the super-clusters formed represent all the steps of an attack present in the set of alerts correlated.

We implemented the first stage of our system using the autoassociator with the CLUSTER-BARRIER algorithm. We found that using the autoassociator as a clustering algorithm gives strong performance results. We also found that the autoassociator has beneficial side effects such as being able to adapt to the network on which it is deployed and generating comparable results between successive runs of the algorithm. We experimented with three different clustering algorithms at the second stage of our system. We found that the EM algorithm produces the best results for this stage.

We tested our complete system against the 1999 DARPA dataset and found that the system performs well on this data. Our system clustered 88.2% of alerts from a set of 500 alerts correctly. We compared our system to a simple clustering algorithm and found

that our system is superior in terms of clustering performance.

Our alert correlation system is significant because it is based on unsupervised machine learning algorithms. Because it is based on machine learning, new information, such as correlation rules, doesn't have to be added to the system as new attacks are discovered and perpetrated, unlike most non-machine learning alert correlation approaches. Because our system is based on unsupervised machine learning rather than supervised machine learning, our system adapts automatically to new network environments and operates with minimal maintenance.

In this thesis we have also explored the autoassociator as a clustering algorithm. In previous research, the autoassociator has been applied to supervised learning problems, but not to unsupervised learning problems. We found that the autoassociator performs well as a clustering algorithm, but that it is necessary to take care in interpretting the autoassociator output. It was necessary in our alert correlation problem to form discrete clusters, so we created the ClusterBarrier algorithm based on single-link clustering.

In conclusion, we have shown the viability of an alert correlation system based on unsupervised machine learning and the autoassociator as a clustering algorithm.

5.1 Future Work

One of the most important future work tasks that we propose is to have the clusters produced by our alert correlation system examined by a seasoned IDS analyst. We believe that helpful hints, if implemented, on the type of practical correlation that is useful in an operational setting would increase the practical relevance of our system.

Also, we would have liked to test the two stages of our system with a more comprehensive set of clustering algorithms. For our work we desire some specific traits of clustering algorithms, as discussed in Section 3.4.3, which limited our choices. For instance, it is useful to evaluate another clustering algorithm that can be trained on unlabelled data, as we determined in Section B.2. Regardless, we could have evaluated other clustering algorithms without this desired property to discover their performance, as we did for the

EM algorithm.

We would have liked to consider some functions of other alert correlation systems, as well. It would have been interesting to investigate alert merging (Cuppens *et al* [36]), alert causation determination (Ning *et al* [45, 53]), attack target analysis (Haines *et al* [50]), and other more advanced features of some alert correlation systems. Alert merging is when the characteristics that define a set of correlated alerts are merged into one reported item. For example, if there are several alerts which encompass a port scan reconnaissance attack, these alerts can be condensed by only reporting a list of the services scanned. Alert merging is particularly relevant to the system we've created.

We would have also liked to experiment with rigourous feature selection at both the first and second stages of our system. We feel that more structured feature selection experiments would have helped increase performance in our system, since features can sometimes decrease clusterer or classifier performance. We also acknowledge that there may be more suitable features that could be constructed to achieve the goals our system; we would like to reconsider feature construction for both stages as well.

We did not use a dedicated set of training data to train the autoassociator or SOMs at the second stage of our system. We would like to experiment with training the clustering algorithm at this stage with data we create. We could run the NAIVECORRELATOR algorithm or the first stage of our system on the whole training set, then use the second stage feature construction without considering that there will be errors in the clusters. We could experiment with training our system on this imperfect data.

It would have been interesting to perform experiments on the autoassociator to examine why it is suitable for some clustering problems (our first stage of correlation, for example) but not for other problems (our second stage of correlation). It is known that different clustering algorithms have different strengths and thus are suitable for different problems and datasets [63, 71, 64]. Therefore, we would like to experiment with clustering several diverse sets of data to understand the strengths of the autoassociator in clustering. We would also like to explore different configurations of the autoassociator to see how they affect the learning capacity of the neural network. For example, it would be interesting to construct and experiment with an autoassociator that has multiple hidden layers.

We have shown that the clustering errors caused by the 40-to-1 reconstruction error formula in the first stage of correlation are significant. But we found that we were able to tune the autoassociator parameters to minimize the effect of the formula. We found that the system performs better as is than if another clustering algorithm is substituted for the ClusterBarrier algorithm part of our system. We would like to correct the 40-to-1 mapping problem if we're able to do so in a way that increases performance.

To ameliorate the 40-to-1 mapping problem we experimented with graph theory-based methods for splitting apart clusters with clustering errors, based on graph theory clustering algorithms [71]. We constructed a graph based on the distance matrices discussed in Section B.3 where each node of the graph is an alert in the cluster and edges are present between nodes with sufficiently short Euclidean distances in the distance matrix. In this method, a cluster was split if the graph we produced was disconnected, where each resulting cluster represented a maximally connected subgraph. We tested this method but nothing came of it because it generated more errors in the overall system than it fixed. But we still believe that this line of research could be fruitful and we'd like to explore it further.

A final future work task we propose is to use non-machine learning methods to increase performance of our system. We would like to combine our system with more simple correlation methods such as the NaiveCorrelator algorithm in Section 3.4.4. We believe that other more complex rule-based systems could increase our performance as well, but creating a hybrid system was beyond the scope of this thesis.

Bibliography

[1] Japkowicz, Nathalie and Smith, Reuben (2005). Autocorrel I: A Neural Network Based Network Event Correlation Approach. (Contractor Report CR 2005-030). DRDC Ottawa. Ottawa, ON.

[2] Japkowicz, Nathalie and Smith, Reuben (2005). Autocorrel II: Unsupervised Network Event Correlation Using Neural Networks. (Contractor Report CR 2005-155). DRDC Ottawa. Ottawa, ON.

[3] Julisch, Klaus and Dacier, Marc (2002). Mining Intrusion Detection Alarms for Actionable Knowledge. In *Proceedings of SIGKDD '02, the 8th International Conference on Knowledge Discovery and Data Mining*, pp. 366–375. Edmonton, Alberta, Canada: ACM Press.

[4] Japkowicz, Nathalie (2001). Supervised Versus Unsupervised Binary-Learning by Feedforward Neural Networks. *Machine Learning*, **42**(1-2), 97–122.

[5] Mitchell, Tom M. (1997). Machine Learning, McGraw-Hill.

[6] Stevens, W. Richard (1994). TCP/IP Illustrated: The Protocols, Vol. 1. Addison-Wesley.

[7] Postel, Jon (1981). RFC 791: Internet Protocol. Obsoletes RFC 760. Status: STANDARD.

[8] Postel, Jon (1981). RFC 793: Transmission Control Protocol. Status: STANDARD.

[9] Northcutt, Stephen (1999). Network Intrusion Detection: An Analyst's Handbook, Indianapolis, IN: New Riders Publishing.

[10] Zimmermann, Hubert (1980). OSI Reference Model—The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, **28**(4).

[11] Postel, Jon (1980). RFC 768: User Datagram Protocol. Status: STANDARD.

[12] Plummer, David C. (1982). RFC 826: Ethernet Address Resolution Protocol. Status: STANDARD.

[13] Finlayson, Ross, Mann, Timothy, Mogul, Jeffery C., and Theimer, Marvin (1984). RFC 903: Reverse Address Resolution Protocol. Status: STANDARD.

[14] Lindqvist, Ulf and Jonsson, Erland (1997). How to Systematically Classify Computer Security Intrusions. In *Proceedings of the 1997 IEEE Symposium on Security & Privacy*, pp. 154–163. IEEE. Oakland, CA: IEEE Computer Society Press, Los Alamitos, CA.

[15] Neumann, Peter G. and Parker, Donn B. (1989). A Summary of Computer Misuse Techniques. In *Proceedings of the 12th National Computer Security Conference*, pp. 396–407. Baltimore, MA.

[16] Landwehr, Carl E., Bull, Alan R., McDermott, John P., and Choi, William S. (1994). A Taxonomy of Computer Program Security Flaws. *ACM Computing Surveys*, **26**(3), 211–254.

[17] Lippmann, Richard, Haines, Joshua W., Fried, David J., Korba, Jonathan, and Das, Kumar (2000). Analysis and Results of the 1999 DARPA Off-Line Intrusion Detection Evaluation. In *Proceedings of the 3rd International Symposium on Recent Advances in Intrusion Detection (RAID 2000), LNCS #1907*, pp. 162–182. Toulouse, France: Springer-Verlag.

[18] Lippmann, Richard, Haines, Joshua W., Fried, David J., Korba, Jonathan, and Das, Kumar (2000). The 1999 DARPA Off-Line Intrusion Detection Evaluation. *Computer Networks*, **34**(4), 579–595.

[19] Lippmann, Richard P., Fried, David J., Graf, Issac, Haines, Joshua W., Kendall, Kristopher R., McClung, David, Weber, Dan, Webster, Seth E., Wyschogrod, Dan, Cunningham, Robert K., and Zissman, Marc A. (2000). Evaluating Intrusion Detection Systems: The 1998 DARPA Off-Line Intrusion Detection Evaluation. In *Proceedings DARPA Information Survivability Conference and Exposition (DISCEX) 2000*, Los Alamitos, CA: IEEE Computer Society Press.

[20] Bejtlich, Richard (2000). Interpreting Network Traffic: A Network Intrusion Detector's Look at Suspicious Events. In *11th FIRST Conference on Computer Security Incident Handling*, Chicago, IL: first.org.

[21] Sheyner, Oleg, Haines, Joshua, Jha, Somesh, Lippmann, Richard, and Wing, Jeannette M. (2002). Automated Generation and Analysis of Attack Graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, IEEE Computer Society.

[22] Cowan, Crispin, Wagle, Perry, Pu, Calton, Beattie, Steve, and Walpole, Jonathan (2000). Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference & Exposition – Volume 2*, pp. 119–129. Orlando, FL: IEEE Press.

[23] Wagner, David, Foster, Jeffrey S., Brewer, Eric A., and Aiken, Alexander (2000). A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2000*, pp. 1–15. San Diego, CA: The Internet Society.

[24] One, Aleph (1996). Smashing the Stack for Fun and Profit. *Phrack*, **7**(49–14). Available from `http://www.phrack.org/phrack/49/P49-14`.

[25] daemon9, route, and infinity (1996). Project Neptune. *Phrack*, **7**(48–13). Available from `http://www.phrack.org/phrack/48/P48-13`.

[26] Debar, Hervé and Wespi, Andreas (2000). Aggregation and Correlation of Intrusion-Detection Alerts. In *Proceedings of the 3rd International Symposium on Recent Advances in Intrusion Detection (RAID 2000), LNCS #1907*, pp. 85–103. Toulouse, France: Springer-Verlag.

[27] Denning, Dorothy E. (1987). An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, **13**(2), 222–232.

[28] Debar, Hervé, Dacier, Marc, and Wespi, Andreas (1999). Towards a Taxonomy of Intrusion-Detection Systems. *Computer Networks*, **31**, 805–822.

[29] Lunt, Teresa F. (1993). A Survey of Intrusion Detection Techniques. *Computer Security*, **12**(4), 405–418.

[30] Axelsson, Stefan (2000). Intrusion-Detection Systems: A Taxonomy and Survey. (Technical Report 99–15). Department of Computer Engineering, Chalmers University of Technology. SE–412 96, Göteborg, Sweden.

[31] Amoroso, Edward (1999). Intrusion Detection, Sparta, NJ: intrusion.net books.

[32] Bace, Rebecca Gurley (2001). Intrusion Detection, First ed. Indianapolis, IN: Macmillan Technical Publishing.

[33] Kemmerer, Richard A. and Vigna, Giovanni (2002). Intrusion Detection: A Brief History and Overview. *IEEE Computer*, pp. 27–30. Special publication on Security and Privacy.

[34] Anderson, Ross (2002). Security in Open versus Closed Systems — The Dance of Boltzmann, Coase and Moore. In *Open Source Software: Economics, Law and Policy Conference*, Toulouse, France.

[35] Cunningham, Robert K., Lippmann, Richard P., Fried, David J., Garfinkel, Simson L., Graf, Issac, Kendall, Kristopher R., Webster, Seth E., Wyschogrod, Dan, and Zissman, Marc A. (1999). Evaluating Intrusion Detection Systems without Attacking your Friends: The 1998 DARPA Intrusion Detection Evaluation. In *Proceedings ID99, Third Conference and Workshop on Intrusion Detection and Response*, San Diego, CA: SANS Institute.

[36] Cuppens, Frédéric and Miége, Alexandre (2002). Alert Correlation in a Cooperative Intrusion Detection Framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pp. 187–200. Oakland, CA: IEEE Computer Society.

[37] Roesch, Martin (1999). Snort—Lightweight Intrusion Detection for Networks. In *Proceedings of LISA '99: 13th Systems Administration Conference*, pp. 229–238. Seattle, Washington: The USENIX Association.

[38] Fan, Wei, Lee, Wenke, Stolfo, Salvatore J., and Miller, Matthew (2000). A Multiple Model Cost-Sensitive Approach for Intrusion Detection. *Department of Computer Science, Columbia University.*

[39] Lee, Wenke and Stolfo, Salvatore J. (2000). A Framework for Constructing Features and Models for Intrusion Detection Systems. *ACM Transactions on Information and System Security*, **3**(4), 227–261.

[40] Eskin, Eleazar, Arnold, Andrew, Prerau, Michael, Portnoy, Leonid, and Stolfo, Sal (2001). A Geometric Framework for Unsupervised Anomaly Detection: Detecting Intrusions in Unlabeled Data. *Department of Computer Science, Columbia University.*

[41] Swets, John A. (1973). The Relative Operating Characteristic in Psychology. *Science*, **182**, 990–1000.

[42] Fawcett, Tom (2003). ROC Graphs: Notes and Practical Considerations for Data Mining Researchers. (Technical Report HPL-2003-4). HP Laboratories. Palo Alto, CA.

[43] Neumann, Peter G. and Porras, Phillip A. (1999). Experience with EMERALD to Date. In *1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California: SRI International Computer Science Lab.

[44] Chyssler, Tobias, Nadjm-Tehrani, Simin, Burschka, Stefan, and Burbeck, Kalle (2004). Alarm Reduction and Correlation in Defence of IP Networks. In *Proceedings of 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'04)*, pp. 229–234. IEEE Computer Society.

[45] Ning, Peng and Cui, Yun (2002). An Intrusion Alert Correlator Based on Prerequisites of Intrusions. (Technical Report TR-2002-01). Department of Computer Science, North Carolina State University. Raleigh, NC.

[46] United States Computer Emergency Readiness Team (Online). Department of Homeland Security. `http://www.us-cert.gov/` (Oct. 20, 2005).

[47] Bugtraq Mailing List (Online). securityfocus.com. `http://www.securityfocus.com/archive/1` (Oct. 20, 2005).

[48] QuIDScor (Online). Qualys Inc.. `http://quidscor.sourceforge.net/` (Sept. 12, 2004).

[49] Morin, Benjamin, Mé, Ludovic, Debar, Hervé, and Ducassé, Mireille (2002). M2D2: A Formal Data Model for IDS Alert Correlation. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002), LNCS #2516*, pp. 115–137. Zurich, Switzerland: Springer-Verlag.

[50] Haines, Joshua, Ryder, Dorene Kewley, Tinnel, Laura, and Taylor, Stephen (2003). Validation of Sensor Alert Correlators. *IEEE Security and Privacy*, pp. 46–56.

[51] Dain, Oliver and Cunningham, Robert K. (2001). Fusing a Heterogeneous Alert Stream into Scenarios. In *Proceedings of the 2001 ACM Workshop on Data Mining for Security Applications*, pp. 1–13. Philadelphia, PA: ACM Press.

[52] Valdes, Alfonso and Skinner, Keith (2001). Probabilistic Alert Correlation. In *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001), LNCS #2212*, pp. 54–68. Davis, CA: Springer-Verlag.

[53] Ning, Peng, Cui, Yun, and Reeves, Douglas S. (2002). Analysing Intensive Intrusion Alerts via Correlation. In *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002), LNCS #2516*, pp. 74–94. Zurich, Switzerland: Springer-Verlag.

[54] Debar, Hervé, Curry, David A., and Feinstein, Benjamin (2004). The Intrusion Detection Message Exchange Format. *IETF Working Group.*

[55] Gorton, Dan (2003). Extending Intrusion Detection with Alert Correlation and Intrusion Tolerance. Licentiate thesis. Chalmers University of Technology.

[56] Hätälä, Antti, Särs, Camillo, Addams-Moring, Ronja, and Virtanen, Teemupekka (2004). Event Data Exchange and Intrusion Alert Correlation in Heterogeneous Networks. In *Proceedings of the 8th Colloquium for Information Systems Security Education (CISSE)*, pp. 84–92. Westpoint, NY: CISSE.

[57] Han, Jiawei, Cai, Yandong, and Cercone, Nick (1993). Data-Driven Discovery of Quantitative Rules in Relational Databases. *IEEE Transactions on Knowledge and Data Engineering*, **5**(1), 29–40.

[58] The DEFCON Dataset (Online). defcon.org. `http://www.defcon.org/` (Not yet checked).

[59] Smith, Reuben, Japkowicz, Nathalie, and Dondo, Maxwell (2005). Clustering Using an Autoassociator: A Case Study in Network Event Correlation. In *Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 613–618. Phoenix, AZ: ACTA Press.

[60] Morin, Benjamin and Debar, Hervé (2003). Correlation of Intrusion Symptoms: An Application of Chronicles. In *Proceedings of the 6th International Symposium on Recent Advances in Intrusion Detection (RAID 2003), LNCS #2820*, Pittsburg, PA: Springer-Verlag.

[61] Danyliw, Roman. ACID: Analysis Console for Intrusion Detections (Online). AIRCERT. `http://www.andrew.cmu.edu/user/rdanyliw/snort/snortacid.html` (Sept. 12, 2004).

[62] Northcutt, Steven and *et al.* SHADOW: Second Heuristic Analysis for Defensive Online Warfare (Online). Naval Surface Warfare Center Dahlgren Laboratory. `http://www.nswc.navy.mil/ISSEC/CID/` (Sept. 12, 2004).

[63] Witten, Ian H. and Frank, Eibe (2000). Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations, The Morgan Kaufmann Series in Data Management Systems. San Francisco, CA: Morgan Kaufmann Publishers.

[64] Jain, Anil K., Murty, M. Narasimha, and Flynn, Patrick J. (1999). Data Clustering: A Review. *ACM Computing Surveys*, **31**(3), 264–323.

[65] Hagan, Martin T., Demuth, Howard B., and Beale, Mark (1996). Neural Network Design, PWS Publishing Company.

[66] Ripley, Brian D. (1996). Pattern Recognition and Neural Networks, First ed. Cambridge: Cambridge University Press.

[67] Lippmann, Richard P. (1987). An Introduction to Computing with Neural Nets. *IEEE ASSP Magazine*, **4**(2), 4–22.

[68] Demuth, Howard B. and Beale, Mark (2001). MATLAB: Neural Network Toolbox, User's Guide v. 4.0, Nantick, MA: The MathWork, Inc.

[69] Zurada, Jacek M. (1992). Introduction to Artificial Neural Systems, New York, NY: West Publishing Company.

[70] Widrow, Bernard and Lehr, Michael A. (1990). 30 Years of Adaptive Neural Networks: Perceptron, Madaline, and Backpropagation. In *IEEE Proceedings*, Vol. 78, pp. 1415–1442.

[71] Duda, Richard O., Hart, Peter E., and Stork, David G. (2001). Pattern Classification, Second ed. New York, NY: John Wiley and Sons.

[72] Dempster, A.P., Laird, N.M., and Rubin, D.B. (1977). Maximum Likelihood from Incoming Data via the EM Algorithm. *J. Royal Stat. Soc., Series B*, **39**(1), 1–36.

[73] Kohonen, Teuvo (1995). Self-Organizing Maps, Vol. 30 of *Springer Series in Information Sciences*. Berlin, Germany: Springer-Verlag. (Second Extended Edition 1997).

[74] Kohonen, Teuvo, Hynninen, Jussi, Kangas, Jari, and Laaksonen, Jorma (1996). SOM_PAK: The Self-Organizing Map Program Package. (Report A31). Laboratory of Computer and Information Science, Helsinki University of Technology. Helsinki, Finland.

[75] Vesanto, Juha and Alhoniemi, Esa (2000). Clustering of the Self-Organizing Map. *IEEE Transactions on Neural Networks*, **11**(3), 586–600.

[76] Tang, Bin, Heywood, Malcolm I., and Shepherd, Michael (2002). Input Partitioning to Mixture of Experts. In *Proceedings of the 2002 International Joint Conference on Neural Networks*, pp. 227–232. Honolulu, HI.

[77] Chiu, S.L. (1994). Fuzzy Model Identification Based on Cluster Estimation. *Journal of Intelligent and Fuzzy Systems*, **2**(3), 267–278.

[78] The incidents.org GIAC Logs (Online). SANS. `http://www.incidents.org/logs/` (Mar. 14, 2004).

[79] Wall, Larry. The Perl programming language (Online). www.perl.com. `http://www.perl.com/` (Sept. 1, 2004).

[80] Hsu, Chih-Wei, Chang, Chih-Chung, and Lin, Chih-Jen (2003). A Practical Guide to Support Vector Classification. *National Taiwan University.* `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`.

[81] McHugh, John (2000). The 1998 Lincoln Laboratory IDS Evaluation — A Critique. In *Proceedings of the 3rd Workshop on Recent Advances in Intrusion Detection (RAID 2000), LNCS #1907*, pp. 145–161. Toulouse, France: Springer-Verlag.

[82] Manning, Christopher D. and Schütze, Hinrich (1999). Foundations of Statistical Natural Language Processing, Cambridge, MA: The MIT Press.

Appendix A

Gold Standard Alerts

This section contains an editted version of the gold standard super-clusters for the incidents.org dataset [78]. This section is editted because it was too long (about 50 pages) to include here otherwise. It has been editted to give an overview of how the gold standard was created.

```
Supercluster 1/25
------------------
Cluster 1/6 (sc:1)
------------------
[**][116:46:1] (snort_decoder) TCP Data Offset is less than 5!  [**]
11/09-20:51:11.676507 62.13.27.29:0->207.166.33.145:0
TCP TTL:234 TOS:0x0 ID:0 IpLen:20 DgmLen:40
*****R** Seq:0x81F9750 Ack:0x81F9750 Win:0x0 TcpLen:0

[**][116:46:1] (snort_decoder) TCP Data Offset is less than 5!  [**]
11/10-01:12:17.866507 172.20.10.199:0->207.166.119.62:0
TCP TTL:235 TOS:0x0 ID:0 IpLen:20 DgmLen:40
*****R** Seq:0xBDD2D468 Ack:0xBDD2D468 Win:0x0 TcpLen:16

[**][116:46:1] (snort_decoder) TCP Data Offset is less than 5!  [**]
11/10-01:28:34.556507 62.13.27.29:0->207.166.78.44:0
TCP TTL:234 TOS:0x0 ID:0 IpLen:20 DgmLen:40 DF
*****R** Seq:0x91D8C02 Ack:0x91D8C02 Win:0x0 TcpLen:12

Cluster 2/6 (sc:1)
------------------
[**][116:46:1] (snort_decoder) TCP Data Offset is less than 5!  [**]
11/12-18:38:17.846507 203.80.239.162:0->207.166.182.137:0
```

```
TCP TTL:107 TOS:0x0 ID:35119 IpLen:20 DgmLen:48 DF
1*UA**** Seq:0x7930005 Ack:0xD80A04D1 Win:0x64BA TcpLen:0 UrgPtr:0x800
...

Supercluster 2/25
------------------
Cluster 1/6 (sc:2)
------------------
[**][119:4:1] (http_inspect) BARE BYTE UNICODE ENCODING [**]
11/14-09:54:38.316507 170.129.50.120:63362->159.153.199.24:80
TCP TTL:125 TOS:0x0 ID:38908 IpLen:20 DgmLen:436 DF
***AP*** Seq:0x99AD8FC7 Ack:0x732FBFCD Win:0x4230 TcpLen:20

[**][119:4:1] (http_inspect) BARE BYTE UNICODE ENCODING [**]
11/14-09:54:47.286507 170.129.50.120:63387->159.153.199.24:80
TCP TTL:125 TOS:0x0 ID:38984 IpLen:20 DgmLen:436 DF
***AP*** Seq:0x99C8B003 Ack:0x733D8EE2 Win:0x4230 TcpLen:20

Cluster 2/6 (sc:2)
------------------
[**][119:4:1] (http_inspect) BARE BYTE UNICODE ENCODING [**]
11/14-13:03:42.666507 170.129.50.120:63598->64.4.22.250:80
TCP TTL:124 TOS:0x0 ID:56064 IpLen:20 DgmLen:932 DF
***AP*** Seq:0x94851318 Ack:0x9AB18054 Win:0x43E1 TcpLen:20

Cluster 3/6 (sc:2)
------------------
[**][119:12:1] (http_inspect) APACHE WHITESPACE (TAB) [**]
11/15-02:46:28.446507 170.129.50.120:64749->216.130.211.11:80
TCP TTL:124 TOS:0x0 ID:40697 IpLen:20 DgmLen:1332 DF
***AP*** Seq:0x1601F507 Ack:0xF2FBCCD5 Win:0x2058 TcpLen:20

[**][119:4:1] (http_inspect) BARE BYTE UNICODE ENCODING [**]
11/15-02:48:02.606507 170.129.50.120:64868->216.130.211.11:80
TCP TTL:124 TOS:0x0 ID:29178 IpLen:20 DgmLen:1332 DF
***AP*** Seq:0x1772D6D5 Ack:0x4E05CFF6 Win:0x2058 TcpLen:20

Cluster 1/5 (sc:3)
------------------
[**][1:184:4] BACKDOOR Q access [**]
11/14-09:29:14.826507 255.255.255.255:31337->170.129.172.186:515
TCP TTL:15 TOS:0x0 ID:0 IpLen:20 DgmLen:43
***A*R** Seq:0x0 Ack:0x0 Win:0x0 TcpLen:20

[**][1:184:4] BACKDOOR Q access [**]
11/14-09:32:53.016507 255.255.255.255:31337->170.129.132.79:515
TCP TTL:15 TOS:0x0 ID:0 IpLen:20 DgmLen:43
***A*R** Seq:0x0 Ack:0x0 Win:0x0 TcpLen:20
```

```
[**][1:184:4] BACKDOOR Q access [**]
11/14-09:49:26.156507 255.255.255.255:31337->170.129.129.188:515
TCP TTL:15 TOS:0x0 ID:0 IpLen:20 DgmLen:43
***A*R** Seq:0x0 Ack:0x0 Win:0x0 TcpLen:20
...

Cluster 2/5 (sc:3)
-------------------
[**][1:184:4] BACKDOOR Q access [**]
11/14-23:47:53.416507 255.255.255.255:31337->170.129.19.28:515
TCP TTL:14 TOS:0x0 ID:0 IpLen:20 DgmLen:43
***A*R** Seq:0x0 Ack:0x0 Win:0x0 TcpLen:20

[**][1:184:4] BACKDOOR Q access [**]
11/14-23:55:50.456507 255.255.255.255:31337->170.129.161.133:515
TCP TTL:14 TOS:0x0 ID:0 IpLen:20 DgmLen:43
***A*R** Seq:0x0 Ack:0x0 Win:0x0 TcpLen:20
...

Supercluster 4/25
------------------
Cluster 1/6 (sc:4)
-------------------
[**][1:628:3] SCAN nmap TCP [**]
11/14-10:10:03.816507 61.218.161.202:80->170.129.19.170:80
TCP TTL:48 TOS:0x0 ID:30084 IpLen:20 DgmLen:40
***A**** Seq:0x134 Ack:0x0 Win:0x578 TcpLen:20

[**][1:628:3] SCAN nmap TCP [**]
11/14-10:10:08.786507 61.218.161.202:80->170.129.19.170:80
TCP TTL:48 TOS:0x0 ID:30366 IpLen:20 DgmLen:40
***A**** Seq:0x198 Ack:0x0 Win:0x578 TcpLen:20

[**][1:628:3] SCAN nmap TCP [**]
11/14-10:10:13.826507 61.218.161.210:80->170.129.19.170:80
TCP TTL:48 TOS:0x0 ID:30662 IpLen:20 DgmLen:40
***A**** Seq:0x20A Ack:0x0 Win:0x578 TcpLen:20
...

Cluster 2/6 (sc:4)
-------------------
[**][1:628:3] SCAN nmap TCP [**]
11/14-12:51:49.906507 61.218.161.202:80->170.129.14.62:80
TCP TTL:48 TOS:0x0 ID:28299 IpLen:20 DgmLen:40
***A**** Seq:0x11B Ack:0x0 Win:0x578 TcpLen:20

[**][1:628:3] SCAN nmap TCP [**]
11/14-12:51:54.916507 61.218.161.202:80->170.129.14.62:80
TCP TTL:48 TOS:0x0 ID:28601 IpLen:20 DgmLen:40
***A**** Seq:0x18D Ack:0x0 Win:0x578 TcpLen:20
```

```
[**][1:628:3] SCAN nmap TCP [**]
11/14-12:51:59.916507 61.218.161.210:80->170.129.14.62:80
TCP TTL:48 TOS:0x0 ID:28911 IpLen:20 DgmLen:40
***A**** Seq:0x209 Ack:0x0 Win:0x578 TcpLen:20
...

Supercluster 8/25
-----------------
Cluster 1/4 (sc:8)
------------------
[**][1:628:3] SCAN nmap TCP [**]
11/14-20:33:44.536507 61.218.15.126:80->170.129.69.49:80
TCP TTL:50 TOS:0x0 ID:9312 IpLen:20 DgmLen:40
***A**** Seq:0x1BF Ack:0x0 Win:0x578 TcpLen:20

[**][1:628:3] SCAN nmap TCP [**]
11/14-20:33:54.676507 61.221.88.198:80->170.129.69.49:80
TCP TTL:50 TOS:0x0 ID:10358 IpLen:20 DgmLen:40
***A**** Seq:0x286 Ack:0x0 Win:0x578 TcpLen:20

[**][1:628:3] SCAN nmap TCP [**]
11/14-20:34:00.056507 192.192.171.251:80->170.129.69.49:80
TCP TTL:44 TOS:0x0 ID:10868 IpLen:20 DgmLen:40
***A**** Seq:0x2EA Ack:0x0 Win:0x578 TcpLen:20
...

Cluster 2/4 (sc:8)
------------------
[**][1:628:3] SCAN nmap TCP [**]
11/15-15:36:25.236507 192.192.171.251:80->170.129.105.7:80
TCP TTL:43 TOS:0x0 ID:10388 IpLen:20 DgmLen:40
***A**** Seq:0x260 Ack:0x0 Win:0x578 TcpLen:20

[**][1:628:3] SCAN nmap TCP [**]
11/15-15:36:17.916507 61.221.88.198:80->170.129.105.7:80
TCP TTL:49 TOS:0x0 ID:9868 IpLen:20 DgmLen:40
***A**** Seq:0x1FC Ack:0x0 Win:0x578 TcpLen:20

[**][1:628:3] SCAN nmap TCP [**]
11/15-15:36:07.886507 61.218.15.126:80->170.129.105.7:80
TCP TTL:49 TOS:0x0 ID:8842 IpLen:20 DgmLen:40
***A**** Seq:0x134 Ack:0x0 Win:0x578 TcpLen:20
...

Cluster 1/1 (sc:15)
-------------------
[**][1:556:5] P2P Outbound GNUTella client request [**]
11/14-15:43:37.096507 170.129.50.120:61121->24.65.114.32:6003
TCP TTL:123 TOS:0x0 ID:22720 IpLen:20 DgmLen:158 DF
***AP*** Seq:0x5A0CA92C Ack:0x53A65413 Win:0x4038 TcpLen:20
```

```
[**][1:556:5] P2P Outbound GNUTella client request [**]
11/14-15:43:37.316507 170.129.50.120:61122->24.65.114.32:6003
TCP TTL:123 TOS:0x0 ID:22766 IpLen:20 DgmLen:62 DF
***AP*** Seq:0x5A129557 Ack:0x53A7A3BA Win:0x4038 TcpLen:20

Supercluster 16/25
------------------
Cluster 1/2 (sc:16)
-------------------
[**][1:620:6] SCAN Proxy Port 8080 attempt [**]
11/14-16:00:56.996507 66.159.18.66:43517->170.129.50.120:8080
TCP TTL:53 TOS:0x0 ID:59575 IpLen:20 DgmLen:60 DF
******S* Seq:0xBED8745 Ack:0x0 Win:0x16D0 TcpLen:40
TCP Options (5) => MSS:1460 SackOK TS:48656370 0 NOP WS:0

[**][1:618:5] SCAN Squid Proxy attempt [**]
11/14-16:00:56.996507 66.159.18.66:43518->170.129.50.120:3128
TCP TTL:53 TOS:0x0 ID:50174 IpLen:20 DgmLen:60 DF
******S* Seq:0xBF3AC0C Ack:0x0 Win:0x16D0 TcpLen:40
TCP Options (5) => MSS:1460 SackOK TS:48656370 0 NOP WS:0
...

Cluster 2/2 (sc:16)
-------------------
[**][1:618:5] SCAN Squid Proxy attempt [**]
11/14-23:32:20.916507 66.159.18.49:55991->170.129.50.120:3128
TCP TTL:52 TOS:0x0 ID:16353 IpLen:20 DgmLen:60 DF
******S* Seq:0xB4E1FC5B Ack:0x0 Win:0x16D0 TcpLen:40
TCP Options (5) => MSS:1460 SackOK TS:51364794 0 NOP WS:0
...

Supercluster 21/25
------------------
Cluster 1/2 (sc:21)
-------------------
[**][1:1390:4] SHELLCODE x86 inc ebx NOOP [**]
11/14-16:10:30.806507 129.118.2.10:57425->170.129.50.120:63414
TCP TTL:51 TOS:0xA0 ID:56986 IpLen:20 DgmLen:1420 DF
***A**** Seq:0x8217BFFC Ack:0x90CF9E29 Win:0x16D0 TcpLen:20

[**][1:648:6] SHELLCODE x86 NOOP [**]
11/14-21:55:36.566507 129.118.2.10:57425->170.129.50.120:63414
TCP TTL:51 TOS:0xA0 ID:46490 IpLen:20 DgmLen:1420 DF
***A**** Seq:0xA074E240 Ack:0x90CF9E29 Win:0x16D0 TcpLen:20

[**][1:648:6] SHELLCODE x86 NOOP [**]
11/14-21:55:36.576507 129.118.2.10:57425->170.129.50.120:63414
TCP TTL:51 TOS:0xA0 ID:46491 IpLen:20 DgmLen:1420 DF
***A**** Seq:0xA074E7A4 Ack:0x90CF9E29 Win:0x16D0 TcpLen:20
...
```

```
Cluster 2/2 (sc:21)
-------------------
[**][1:1390:4] SHELLCODE x86 inc ebx NOOP [**]
11/14-21:56:03.856507 129.118.2.10:57425->170.129.50.120:63414
TCP TTL:51 TOS:0x0 ID:48298 IpLen:20 DgmLen:1420 DF
***AP*** Seq:0xA09AF480 Ack:0x90CF9E29 Win:0x16D0 TcpLen:20

[**][1:1390:4] SHELLCODE x86 inc ebx NOOP [**]
11/14-21:56:03.906507 129.118.2.10:57425->170.129.50.120:63414
TCP TTL:51 TOS:0xA0 ID:48302 IpLen:20 DgmLen:1420 DF
***AP*** Seq:0xA09B0A10 Ack:0x90CF9E29 Win:0x16D0 TcpLen:20
...

Supercluster 22/25
------------------
Cluster 1/13 (sc:22)
-------------------
[**][1:524:7] BAD-TRAFFIC tcp port 0 traffic [**]
11/15-07:36:26.406507 211.47.255.24:41866->170.129.195.40:0
TCP TTL:46 TOS:0x0 ID:0 IpLen:20 DgmLen:52 DF
******S* Seq:0xD8010CF5 Ack:0x0 Win:0x16D0 TcpLen:32
TCP Options (6) => MSS:1460 NOP NOP SackOK NOP WS:0

[**][1:524:7] BAD-TRAFFIC tcp port 0 traffic [**]
11/15-07:36:29.296507 211.47.255.24:41866->170.129.195.40:0
TCP TTL:46 TOS:0x0 ID:0 IpLen:20 DgmLen:52 DF
******S* Seq:0xD8010CF5 Ack:0x0 Win:0x16D0 TcpLen:32
TCP Options (6) => MSS:1460 NOP NOP SackOK NOP WS:0
...

Cluster 2/13 (sc:22)
-------------------
[**][1:524:7] BAD-TRAFFIC tcp port 0 traffic [**]
11/15-15:09:56.016507 211.47.255.23:47620->170.129.23.96:0
TCP TTL:46 TOS:0x0 ID:0 IpLen:20 DgmLen:52 DF
******S* Seq:0x88C07AEB Ack:0x0 Win:0x16D0 TcpLen:32
TCP Options (6) => MSS:1460 NOP NOP SackOK NOP WS:0

[**][1:524:7] BAD-TRAFFIC tcp port 0 traffic [**]
11/15-15:10:02.006507 211.47.255.23:47620->170.129.23.96:0
TCP TTL:46 TOS:0x0 ID:0 IpLen:20 DgmLen:52 DF
******S* Seq:0x88C07AEB Ack:0x0 Win:0x16D0 TcpLen:32
TCP Options (6) => MSS:1460 NOP NOP SackOK NOP WS:0 ...

Supercluster 23/25
------------------
Cluster 1/3 (sc:23)
-------------------
[**][1:527:4] BAD-TRAFFIC same SRC/DST [**]
```

```
11/14-22:36:45.306507 170.129.215.99->170.129.215.99
IGMP TTL:47 TOS:0x0 ID:0 IpLen:20 DgmLen:28

[**][1:527:4] BAD-TRAFFIC same SRC/DST [**]
11/14-22:36:45.306507 170.129.215.104->170.129.215.104
IGMP TTL:47 TOS:0x0 ID:0 IpLen:20 DgmLen:28
...

Cluster 2/3 (sc:23)
-------------------
[**][1:527:4] BAD-TRAFFIC same SRC/DST [**]
11/16-03:26:16.456507 170.129.71.37->170.129.71.37
IGMP TTL:46 TOS:0x0 ID:0 IpLen:20 DgmLen:28

[**][1:527:4] BAD-TRAFFIC same SRC/DST [**]
11/16-03:26:16.456507 170.129.71.42->170.129.71.42
IGMP TTL:46 TOS:0x0 ID:0 IpLen:20 DgmLen:28
...

Supercluster 24/25
------------------
Cluster 1/3 (sc:24)
-------------------
[**][1:523:4] BAD-TRAFFIC ip reserved bit set [**]
11/14-11:21:09.916507 200.200.200.1->170.129.211.200
TCP TTL:242 TOS:0x0 ID:0 IpLen:20 DgmLen:40 RB
Frag Offset:0x0864 Frag Size:0x0014

[**][1:523:4] BAD-TRAFFIC ip reserved bit set [**]
11/14-14:37:18.296507 200.200.200.1->170.129.2.16
TCP TTL:242 TOS:0x0 ID:0 IpLen:20 DgmLen:40 RB
Frag Offset:0x0864 Frag Size:0x0014
...
```

Appendix B

Additional Experiments

We include three additional experiments with the first stage of correlation of our system as this appendix. We include these results as an appendix instead of in Section 4.2 because they are not central to the discussion of this thesis. The properties of the autoassociator explored in these experiments are referred to throughout the thesis, but we do not consider them critical to the success of the system.

In Section B.3 we explore the problem of the 40-to-1 mapping discussed in Section 3.4.1. In Section B.1 we explore the effect of the 10 000 unlabelled training alerts on the performance of our system. That is, we try to determine if using the large unlabelled training set affects performance by substituting the large training set for a smaller, different set. In Section B.2 we present an experiment that shows the role of the 10 000 unlabelled training alerts in making the reconstruction error values produced by the autoassociator more stable numerically.

B.1 Autoassociator Training

We wanted to investigate the effect of training the autoassociator in the first stage of our system. In particular, we were interested in understanding how training the autoassociator might affect the performance of the clustering algorithm. To determine the effects

of training on performance, we propose an experiment to analyze the impact of using a large number of unlabelled alert data to train our system. To do this analysis we compare the results of the first stage when the autoassociator is trained on 10 000 unlabelled alert data against the same system trained on the set of alerts to be clustered before they are labelled.

As discussed in Section 2.4, training is a common process for most data mining and machine learning applications. Training in data mining or machine learning represents the learning part of the process; this idea is based on the fact that, in most applications, past decisions are good predictors of future decisions. But in the domain of alert correlation, this assumption is violated. Julisch *et al* [3] explicitly mention the idea that traditional data mining training may not be valuable in the alert correlation domain. McHugh [81] mentions the fact that each site that runs an IDS will have a unique distribution of alarm types. Julisch *et al* strengthen this by showing that even the distribution of alerts for a given network changes from month to month.

The systems of Julisch *et al* and Dain *et al* [51] represent the most serious attempts by researchers to apply machine learning algorithms to the domain of alert correlation. Both of these papers rely to some extent on supervised training using labelled datasets, and both papers present novel ideas for doing so. For our system we wish to investigate whether our type of unsupervised training is necessary for reasonable performance of a correlation system.

We investigated training by considering a variation of the first stage of our system implemented with an autoassociator that is not trained using the 10 000 unlabelled data instances. For this variation we trained the autoassociator on the 100 alerts of the evaluation data to be clustered without using their labels, then reconstructed the same data on the neural network after training had completed. What we expected is that the autoassociator trained in this way could find more subtle numerical differences between alerts, since it had intimately learned the data. We also expected that the reconstruction errors for the data items will be less stable, but we explored this hypothesis in Section B.2. We call this variation of the trained network the "circularly-trained autoassociator" for easy reference later.

This experiment will show the effect of the 10 000 item training dataset on the performance of the first stage of our system. We compare the new system variation to the results of our first stage as reported in Section 3.2 and Section 3.4.1 on the basis of performance.

We acknowledge that a system trained on the data to be clustered is impractical because the autoassociator would have to be retrained for every new window of 100 alerts. This variation of the system was intended to explore the effect of the 10 000 unlabelled training data. This variation was created for the purposes of comparison and was not intended to be a viable system. We only imagined one other possible system variation for comparison with our first stage to explore the effect of the 10 000 unlabelled training data. This other system variation would have been a completely untrained autoassociator. Since neural networks are supposed to be trained, we discounted this variation because the performance of this system would have been unfairly poor.

For this experiment, we examine the autoassociator with 8, 16, and 32 hidden units. We evaluate the performance against the evaluation data at 500, 2500 and 5000 epochs. We used self-organizing maps, rather than the ClusterBarrier algorithm, to form discrete clusters with the autoassociator output data. The self-organizing maps are trained for 1000 epochs on the same data that they cluster. The self-organizing maps are configured in a 4×6 lattice.

In this section we present results for both the incidents.org and 1999 DARPA datasets. The results of the different algorithm configurations are comparable only if the evaluation dataset is an invariant because different sets of data have different innate characteristics. We see the statistics for the 10 000 alert training data in Table B.1[1] for the incidents.org data, and we see the statistics for the same experiment with the 1999 DARPA dataset in Table B.2[2]. The results for the experiment that doesn't include unlabelled training data on the incidents.org dataset are in Table B.3 and on the 1999 DARPA dataset in Table B.4.

We see from Table B.5 that the "circularly-trained autoassociator" system produced

[1]The results in Table B.1 are a reprint of the results in Table 4.3.
[2]The results in Table B.2 are a reprint of the results in Table 4.7.

Table B.1: Results with incidents.org alerts of the autoassociator trained on 10 000 data

	Epochs=500			Epochs=2500			Epochs=5000		
Hidden Units	TE	CE	SE	TE	CE	SE	TE	CE	SE
8	34	6	28	33	4	29	31	3	28
16	33	4	29	33	4	29	34	4	30
32	29	4	25	28	2	26	28	5	23

Table B.2: Results with DARPA alerts of the autoassociator trained on 10 000 data

	Epochs=500			Epochs=2500			Epochs=5000		
Hidden Units	TE	CE	SE	TE	CE	SE	TE	CE	SE
8	62	4	58	61	4	57	57	7	50
16	61	3	58	61	2	59	59	3	56
32	58	6	52	60	9	51	59	2	57

Table B.3: Results with incidents.org alerts of autoassociator trained on evaluation data

	Epochs=500			Epochs=2500			Epochs=5000		
Hidden Units	TE	CE	SE	TE	CE	SE	TE	CE	SE
8	33	5	28	34	6	28	34	4	30
16	29	2	27	31	2	29	28	2	26
32	27	1	26	32	5	27	28	2	26

Table B.4: Results with DARPA alerts of autoassociator trained on evaluation data

	Epochs=500			Epochs=2500			Epochs=5000		
Hidden Units	TE	CE	SE	TE	CE	SE	TE	CE	SE
8	63	3	60	60	6	54	61	3	58
16	61	8	53	58	2	56	63	9	54
32	62	3	59	64	14	50	65	6	59

similar averages to the averages produced by the equivalent systems using the 10 000 unlabelled training data. (For the statistics in this table, mean averages are typeset in normal font and standard deviations are *italicized.*)

Table B.5: Average errors for both datasets

	Average Errors		
Dataset	TE	CE	SE
incidents.org (10 000 item training set)	31.5 (*6.1*)	4	27.5
1999 DARPA (10 000 item training set)	59.5 (*2.5*)	5	54.5
incidents.org (evaluation training set)	30.7 (*7.5*)	3.2	27.4
1999 DARPA (evaluation training set)	61.9 (*4.7*)	6	55.9

From Table B.3 we see that the autoassociator trained on the 10 000 training data using the SOMs for clustering produced an error rate of 31.5% on average, whereas the "circularly-trained autoassociator" version produced an error rate of 30.7% on average. Also, from Table B.4 we see that the system trained with 10 000 alerts produced an error rate of 59.5% on average, and the comparable "circularly-trained autoassociator" system produced an error rate of 61.9% on average.

This experiment shows that the system does not benefit and is not impaired by the use of the unlabelled training data. The fact that the averages are so close for the two datasets shows that the training data does not affect performance. Note that we cannot conclude anything about the stability of the reconstruction error values from this experiment, since this experiment measured clustering performance. We explore stability in depth in Section B.2.

B.2 Autoassociator Reconstruction Error Stability

In this experiment we show that the reconstruction errors produced by the autoassociator change less significantly if we train the autoassociator with a training set of 10 000 unlabelled alerts. Minimizing change in the reconstruction error of a constant alert in different contexts is important because it allows the IDS analyst to find correlation between different windows of alerts more easily. If the reconstruction error of a given alert

does not change significantly in different contexts of clustering then the reconstruction error of that alert can be seen as a canonical numbering of that alert.

The second correlation stage, described in Section 3.3, capitalizes on this trait of stability in the first stage. Stability in reconstruction errors is important because if reconstruction error values for a set of similar alerts change insignificantly, then alerts are comparable between test sets, which bolsters the second stage of our correlation system. This clustering algorithm trait is discussed in Section 3.4.3 as well.

We designed an experiment to test the stability of a set alerts. For this experiment we used parameter values that we have found to perform well. Namely, we created an autoassociator with 64 hidden units and trained the autoassociator for 500 epochs. We used the 10 000 unlabelled alert training set from the incidents.org dataset. We contrast the output of this system with the reconstruction errors of a system variation that does not use the 10 000 training data, a system that both trains on and clusters the evaluation data. The latter system was presented in Section B.1 as the "circularly-trained autoassociator" system.

For this experiment we chose a constant set of 100 alerts for which we report the reconstruction errors. This set of 100 alerts contains 50 "Scan Proxy Port 8080" alerts and 50 "Short UDP Packet" alerts. For this set of 100 alerts we report on the reconstruction errors generated when combined with a greater set of 400 other alerts. For one experiment this additional set of 400 alerts is comprised of "Scan Proxy Port 8080" alerts and for another experiment the additional set of 400 alerts is comprised of "Short UDP Packet" alerts.

We expected the reconstruction errors of the 100 evaluation alerts would exhibit less stability if the autoassociator is trained with the evaluation set itself than if the autoassociator is trained with the 10 000 alert training set. Changing the distribution of the evaluation set showed that using the evaluation set as training data leads to unstable reconstruction errors.

To explore the impact on reconstruction errors we used our system to determine the reconstruction errors on the two datasets, in particular the constant set of 100 alerts present in the two datasets. As mentioned both evaluation datasets have 500 alerts, but

one dataset has 450 "Scan Proxy Port 8080" alerts with only 50 "Short UDP Packet" alerts, and the other dataset of 500 alerts has 450 of the latter type of alert and only 50 of the former. We report on the difference in reconstruction error values for 100 of the 500 alerts from the two datasets. The 100 alerts examined are the same in both datasets. See Figure B.1 for a graph of the results.

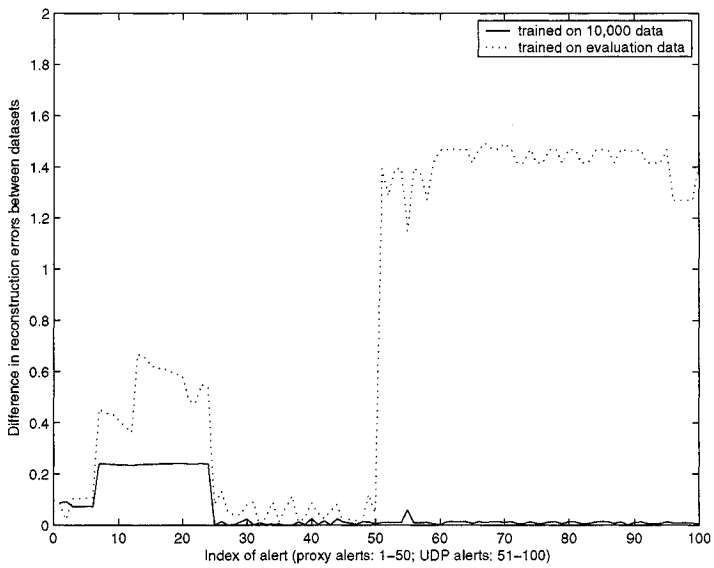

Figure B.1: Impact of using a training set on the stability of reconstruction errors

To measure the stability of the reconstruction errors we determined the reconstruction error values for both of the datasets for each different training set, then took the difference of the reconstruction errors for the data items that were in both datasets. The results of the experiment for the autoassociator trained with the 10 000 training set alerts are the solid line in Figure B.1. The results for the autoassociator trained on the evaluation set are the dotted line in the same figure.

As presented in the graph, the differences in reconstruction errors for the autoassociator trained on the evaluation set are greater than those for the autoassociator trained with the 10 000 alert training set. This is especially true of the "Short UDP Packet" alerts which are represented by the high plateau at the right side of the graph. The "Scan Proxy Port 8080" alerts from the set that the autoassociator trained with the 10 000

alert training set displayed more stable reconstruction errors too. The differences for these alerts are not as dramatic, but it is still clear that the autoassociator trained with 10 000 alerts is uniformly more stable than if trained on the evaluation data.

From the previous experiment we can conclude that the reconstruction error values produced by an autoassociator trained on the 10 000 alert training set are much more stable than if the autoassociator is not trained on this data and instead trained on the evaluation data.

B.3 Feature Mapping Problem

To prove conclusively that the 40-to-1 mapping is a significant source of errors, we devised and ran an experiment. To perform the experiment we analyzed the errors produced by our system and determined whether they were caused by the 40-to-1 feature mapping. We used an evaluation dataset of 100 alerts for this experiment.

The 40-to-1 mapping problem occurs because of the reconstruction error formula used in the interpretation of the autoassociator outputs. For each alert, the reconstruction error formula computes a single value from the 40 autoassociator outputs, hence the "40-to-1 mapping problem" name. The feature mapping problem is explained in detail in Section 3.4.1.

We were most concerned with the errors where intuitively dissimilar alerts are grouped together. We call this type of errors clustering errors (see Section 4.1.3) because they indicate the problem of the system finding correlation where there is none.

We hypothesized that some of the clustering errors we had seen previously in our tests were caused by the 40-to-1 mapping, but we did not know the percentage of clustering errors. To calculate the percentage of clustering errors caused by the mapping problem, we needed to determine whether each individual clustering error was caused by the mapping problem, then simply count the number of clustering errors caused by the mapping problem and compare that to the total number of clustering errors.

We note that the reconstruction error formula can't cause any separation errors, the

other type of error encountered in our clustering system. Separation errors are errors seen when a gold standard cluster is split into two or more separate clusters in our system output, so our deterministic 40-to-1 mapping, defined by the reconstruction error formula, would always map numerically similar data items to similar domain values. The problem with the reconstruction error formula is that it can map dissimilar data items to similar domain values, and the effects of this are only clustering errors.

To determine whether a particular clustering error is caused by the 40-to-1 mapping we needed to look at the values which compose the computed reconstruction error values. Since the reconstruction error E_x of data item x, from Equation 2.12, is the output of the 40-to-1 mapping, we looked the 40 inputs to this formula, $d_{x_i} = x_i - f_i(x)$. The clusters in our system are formed by examining the differences in behaviour between mappings of the components of $d_x = [d_{x_1}, \ldots, d_{x_n}]$ values for different data items x. That is, two data items x and y should only be clustered together if the values d_{x_i} and d_{y_i} are reasonably similar for all i. A suitable test of similarity for high-dimensional vectors helped us test if two data items were clustered together correctly.

There exist a number of vector similarity measures to quantify this similarity (Manning and Schütze [82]), for example *cosine similarity* and *Euclidean distance*, also known as *geometric distance*. We chose to use Euclidean distance because the measure is well-known and the results are easily interpretable. Manning and Schütze define Euclidean distance between vectors x and y as follows:

$$|x - y| = \sqrt{\sum_{i=1}^{n} [x_i - y_i]^2} \tag{B.1}$$

We used the Euclidean distance measure to measure the distances between all the combinations of two vectors in clusters that contain clustering errors. We computed the distance $|d_x - d_y|$ for all such pairs and found that the distances are significant in a cluster between intuitively anomalous alerts and the rest of the cluster.

When running our experiment we found four clusters with clustering error, meaning that there were four system-produced clusters that contained alerts from two or more

gold standard clusters. Clusters 4, 6, 8 and 22 were the four system-produced clusters containing clustering errors. We discuss these clusters below.

Table B.6: Euclidean distances between alerts in cluster 4

	26	27	82	81	80	**7**	79	78	77	76
26	0.0000	0.0113	0.0299	0.0357	0.0242	**2.8747**	0.0438	0.0542	0.0639	0.0739
27	0.0113	0.0000	0.0188	0.0258	0.0262	**2.8737**	0.0436	0.0534	0.0627	0.0725
82	0.0299	0.0188	0.0000	0.0108	0.0348	**2.8714**	0.0454	0.0531	0.0611	0.0698
81	0.0357	0.0258	0.0108	0.0000	0.0328	**2.8689**	0.0387	0.0451	0.0523	0.0604
80	0.0242	0.0262	0.0348	0.0328	0.0000	**2.8693**	0.0200	0.0304	0.0401	0.0502
7	**2.8747**	**2.8737**	**2.8714**	**2.8689**	**2.8693**	**0.0000**	**2.8647**	**2.8625**	**2.8603**	**2.8581**
79	0.0438	0.0436	0.0454	0.0387	0.0200	**2.8647**	0.0000	0.0104	0.0202	0.0302
78	0.0542	0.0534	0.0531	0.0451	0.0304	**2.8625**	0.0104	0.0000	0.0098	0.0198
77	0.0639	0.0627	0.0611	0.0523	0.0401	**2.8603**	0.0202	0.0098	0.0000	0.0101
76	0.0739	0.0725	0.0698	0.0604	0.0502	**2.8581**	0.0302	0.0198	0.0101	0.0000

Table B.6 shows a matrix of distances between the alerts of cluster 4. Each of the alerts in the dataset is assigned a unique index number for easy reference, and the numbers of the alerts in this cluster are 26, 27, 82, 81, 80, 7, 79, 78, 77, and 76. The anomalous alert is alert 7, which is an alert of type "Short UDP Packet". (The distances in Table B.6 that involve the anomalous alert, alert 7, are typeset in **bold**.) We include the Snort representation of this alert:

```
[**][116:97:1] Short UDP packet, length field > payload length [**]
11/11-13:29:54.796507 211.194.68.39:0->207.166.72.218:0
UDP TTL:109 TOS:0x0 ID:2062 IpLen:20 DgmLen:78
Len:129
```

The rest of the alerts in this cluster are "Scan nmap TCP" alerts, which are correctly clustered together. We include the Snort representation of alert 26, a sample of this type of alert:

```
[**][1:628:3] SCAN nmap TCP [**]
11/14-12:02:45.976507 167.79.91.3:80->170.129.50.122:53
TCP TTL:49 TOS:0x0 ID:11661 IpLen:20 DgmLen:40
**A**** Seq:0x2A5 Ack:0x0 Win:0x578 TcpLen:20
```

From Table B.6 you can see that the Euclidean distance between alert 7 and the rest of

the alerts in the cluster ranges between 2.85 and 2.88, whereas the distance between any two of the rest is at most 0.07.

Alert 7 is clearly numerically anomalous in this cluster, yet our CLUSTERBARRIER algorithm has clustered it with the other alerts because the reconstruction error value of alert 7 is similar to the reconstruction error values of the other alerts in the cluster. This shows that the reconstruction error formula is responsible for the clustering error in cluster 4.

Table B.7: Euclidean distances between alerts in cluster 6

	28	29	30	31	**59**	32	33	**58**
28	0.0000	0.0114	0.0224	0.0342	**3.4449**	0.0655	0.0762	**3.4049**
29	0.0114	0.0000	0.0110	0.0228	**3.4431**	0.0543	0.0649	**3.4027**
30	0.0224	0.0110	0.0000	0.0118	**3.4415**	0.0436	0.0542	**3.4006**
31	0.0342	0.0228	0.0118	0.0000	**3.4398**	0.0324	0.0427	**3.3984**
59	**3.4449**	**3.4431**	**3.4415**	**3.4398**	**0.0000**	**3.4353**	**3.4339**	**0.1864**
32	0.0655	0.0543	0.0436	0.0324	**3.4353**	0.0000	0.0108	**3.3927**
33	0.0762	0.0649	0.0542	0.0427	**3.4339**	0.0108	0.0000	**3.3909**
58	**3.4049**	**3.4027**	**3.4006**	**3.3984**	**0.1864**	**3.3927**	**3.3909**	**0.0000**

The matrix in Table B.7 shows the distances between the alerts of cluster 6. (Again, the distances involving anomalous alerts in this table are typeset in **bold**.) In this cluster there are two anomalous alerts that contribute to the clustering errors. The alerts 58 and 59 are anomalous; alert 58 is of type "SCAN Proxy Port 8080 attempt" and alert 59 is of type "SCAN Squid Proxy attempt". The other alerts in this cluster are "Scan nmap TCP" alerts, similar to alert 26 that was previously shown. Here are the Snort representations of alerts 58 and 59:

```
[**][1:620:6] SCAN Proxy Port 8080 attempt [**]
11/14-16:00:56.996507 66.159.18.66:43517->170.129.50.120:8080
TCP TTL:53 TOS:0x0 ID:59575 IpLen:20 DgmLen:60 DF
*****S* Seq:0xBED8745 Ack:0x0 Win:0x16D0 TcpLen:40
TCP Options (5) => MSS:1460 SackOK TS:48656370 0 NOP WS:0

[**][1:618:5] SCAN Squid Proxy attempt [**]
11/14-16:00:56.996507 66.159.18.66:43518->170.129.50.120:3128
```

```
TCP TTL:53 TOS:0x0 ID:50174 IpLen:20 DgmLen:60 DF
*****S* Seq:0xBF3AC0C Ack:0x0 Win:0x16D0 TcpLen:40
TCP Options (5) => MSS:1460 SackOK TS:48656370 0 NOP WS:0
```

This cluster is interesting because the two anomalous alerts should actually be correlated together, but not with the rest of the alerts in the cluster. There should be a division placed between the alerts 58 and 59, and the other alerts in the cluster. We know that alert 58 should be correlated with alert 59 because the Euclidean distance between the pair of alerts is about 0.19. In contrast, the distance between the other alerts in the cluster and alerts 58 or 59 is at least 3.40. The distance between any of pair of the other alerts is at most 0.08, showing that they are correctly clustered together. We can conclude again that the two anomalous alerts are responsible for the clustering errors in cluster 6.

Table B.8: Euclidean distances between alerts in cluster 8

	24	49	57	73	**60**
24	0.0000	0.0000	0.0000	0.0000	**3.7026**
49	0.0000	0.0000	0.0000	0.0000	**3.7026**
57	0.0000	0.0000	0.0000	0.0000	**3.7026**
73	0.0000	0.0000	0.0000	0.0000	**3.7026**
60	**3.7026**	**3.7026**	**3.7026**	**3.7026**	**0.0000**

The matrix of real values in Table B.8 shows the distances between the alerts of cluster 8. There is one anomalous alert in this cluster, alert 60, of type "SCAN SOCKS proxy attempt". The other alerts in cluster 8 are of type "BAD-TRAFFIC ip reserved bit set". We give the Snort representation of alert 60:

```
[**][1:615:5] SCAN SOCKS Proxy attempt [**]
11/14-16:00:56.996507 66.159.18.66:43520->170.129.50.120:1080
TCP TTL:53 TOS:0x0 ID:4253 IpLen:20 DgmLen:60 DF
*****S* Seq:0xB9A7F6F Ack:0x0 Win:0x16D0 TcpLen:40
TCP Options (5) => MSS:1460 SackOK TS:48656370 0 NOP WS:0
```

And we also present alert 24, typical of the other alerts in the cluster:

```
[**][1:523:4] BAD-TRAFFIC ip reserved bit set [**]
```

```
11/14-11:21:09.916507 200.200.200.1->170.129.211.200
TCP TTL:242 TOS:0x0 ID:0 IpLen:20 DgmLen:40 RB
Frag Offset:0x0864 Frag Size:0x0014
```

This cluster is interesting because it is an extreme example of how the mapping problem caused by the reconstruction error formula generates clustering errors in our system. The Euclidean distance between 60 and all the other alerts is 3.7026 and the Euclidean distance between any pair of the other alerts is 0. The distance is exactly 0 because the "BAD-TRAFFIC ip reserved bit set" alerts in this cluster all have exactly the same numerical representation. This is a strong indicator that these alerts should be clustered together, and it is obvious that alert 60 is anomalous to this set.

Table B.9: Euclidean distances between alerts in cluster 22

	40	9
40	**0.0000**	**3.0659**
9	**3.0659**	0.0000

The last cluster containing clustering errors is cluster 22. This cluster contains only two alerts, and the Euclidean distance between these two alerts is 3.0659. (We include this value as a matrix in Table B.9 to be consistent with the analysis performed of the other clusters.) Alert 40 is a "BARE BYTE UNICODE ENCODING" alert and alert 9 is a "TCP Data Offset is less than 5" alert. We include the Snort representations of these two alerts:

```
[**][116:46:1] TCP Data Offset is less than 5!  [**]
11/12-18:38:17.846507 203.80.239.162:0->207.166.182.137:0
TCP TTL:107 TOS:0x0 ID:35119 IpLen:20 DgmLen:48 DF
1*UA**** Seq:0x7930005 Ack:0xD80A04D1 Win:0x64BA TcpLen:0 UrgPtr:0x800
```

```
[**][119:4:1] (http_inspect) BARE BYTE UNICODE ENCODING [**]
11/14-13:03:42.666507 170.129.50.120:63598->64.4.22.250:80
TCP TTL:124 TOS:0x0 ID:56064 IpLen:20 DgmLen:932 DF
**AP*** Seq:0x94851318 Ack:0x9AB18054 Win:0x43E1 TcpLen:20
```

It should be clear that these two alerts being clustered together constitute a clustering

error. As mentioned, the Euclidean distance is 3.0659. Clearly these alerts lay some distance apart in 40-dimensional space, but the CLUSTERBARRIER algorithm has clustered them together because their reconstruction error values are similar. This small cluster is a simple example of how the reconstruction error formula causes clustering errors.

In conclusion, all of the clustering errors in this experiment were caused by the reconstruction error formula incorrectly mapping from a 40-dimensional space to a single dimension.

Clearly the errors caused by this 40-to-1 mapping are significant. While the sample size of this experiment is small, we believe it is representative, and that the reconstruction error formula accounts for a significant portion of the clustering errors produced by our system. To correct the 40-to-1 mapping problem we considered using a different clustering algorithm (see Section 4.2.2) but we found that other clustering algorithms performed even poorer than the autoassociator without the 40-to-1 problem fixed. Other solutions to this problem are proposed in Section 5.1, the section on future work.