2000

Intelligent multi-agent system for intrusion detection and countermeasures

Guy Gary Helmer

Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation

http://lib.dr.iastate.edu/rtd/12328

This Dissertation is brought to you for free and open access by Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
NOTE TO USERS

This reproduction is the best copy available.

UMI
Intelligent multi-agent system for intrusion detection and countermeasures

by

Guy Gary Helmer

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Major Professor: Johnny S. K. Wong

Iowa State University
Ames, Iowa
2000
This is to certify that the Doctoral dissertation of
Guy Gary Helmer
has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Committee Member
Signature was redacted for privacy.
Committee Member
Signature was redacted for privacy.
Committee Member
Signature was redacted for privacy.

Major Professor
Signature was redacted for privacy.

For the Graduate College
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td></td>
<td>xii</td>
</tr>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Computer Security and Intrusion Detection</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1</td>
<td>Security Policy</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Misuse Detection</td>
<td>2</td>
</tr>
<tr>
<td>1.1.3</td>
<td>Algorithms for Misuse Detection</td>
<td>2</td>
</tr>
<tr>
<td>1.1.4</td>
<td>Anomaly Detection</td>
<td>3</td>
</tr>
<tr>
<td>1.1.5</td>
<td>Algorithms for Anomaly Detection</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Audit Records</td>
<td>5</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Audit Data Attributes</td>
<td>5</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Data Sources in Systems and Networks</td>
<td>6</td>
</tr>
<tr>
<td>1.2.3</td>
<td>Means of Obtaining Audit Data</td>
<td>7</td>
</tr>
<tr>
<td>1.2.4</td>
<td>Relationship of Available Data Sources to Graph Model</td>
<td>10</td>
</tr>
<tr>
<td>1.3</td>
<td>Intrusion Taxonomy</td>
<td>11</td>
</tr>
<tr>
<td>1.4</td>
<td>Descriptions of Classified Intrusions</td>
<td>13</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Network Protocol Intrusions</td>
<td>13</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Host Intrusions</td>
<td>20</td>
</tr>
<tr>
<td>1.4.3</td>
<td>Network Application Intrusions</td>
<td>24</td>
</tr>
<tr>
<td>1.5</td>
<td>Classifications using Intrusion Taxonomy</td>
<td>27</td>
</tr>
<tr>
<td>1.6</td>
<td>Glossary</td>
<td>27</td>
</tr>
<tr>
<td>1.7</td>
<td>Thesis Statement and Outline</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>RELATED WORK</td>
<td>34</td>
</tr>
<tr>
<td>2.1</td>
<td>Intrusion Detection System Architectures</td>
<td>34</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Common Intrusion Detection Framework</td>
<td>34</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Distributed Intrusion Detection System</td>
<td>34</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1.1 Sample syslog entries .................................................. 8
Figure 1.2 Diagram of an FTP Bounce Attack ................................. 10
Figure 1.3 TCP connection establishment diagram .......................... 16

Figure 2.1 Beginning of a worm graph, and the graph after the worm has spread [123] ... 35

Figure 3.1 Original architecture of the Mobile Agents Intrusion Detection System ........................ 41
Figure 3.2 Enterprise data warehouse for intrusion detection ............... 42
Figure 3.3 Facets for the low-level agents ...................................... 48
Figure 3.4 A portion of the intrusion detection system with facets .......... 53

Figure 5.1 Relevant fault tree symbols .......................................... 71
Figure 5.2 Reconnaissance fault tree ............................................ 74
Figure 5.3 Vulnerability identification fault tree ............................... 76
Figure 5.4 Penetration fault tree .................................................. 77
Figure 5.5 Penetration fault tree: Using buffer overflows in network daemons ................. 78
Figure 5.6 Penetration fault tree: Gaining access by modifying configuration files .......... 79
Figure 5.7 Penetration fault tree: Gaining access through abuse of authentication methods .......... 80
Figure 5.8 Control fault tree ....................................................... 81
Figure 5.9 Embedding fault tree .................................................. 83
Figure 5.10 Attack relay fault tree ............................................... 85

Figure 6.1 CPX for the entire FTP Bounce intrusion (ftp-bounce-hier-cpn) ............... 99
Figure 6.2 CPX for the reconnaissance stage of the FTP Bounce intrusion (ftp-bounce-scan-cpn) .................................................. 100
Figure 6.3 CPX for scanning for a host providing the FTP service (ftp-bounce-scan-ftp-cpn) 100
Figure 6.4 CPX for scanning for a host providing the RSH service (ftp-bounce-scan-rsh-cpn) 101
Figure 6.5 CPN for host scanning detector (host-scan-cpn) ........................................ 101
Figure 6.6 CPN for port scanning detector (port-scan-cpn) ........................................ 102
Figure 6.7 Simple CPN for identifying FTP server vulnerable to FTP Bounce (ftp-bounce-vuln-cpn) ................................................................. 102
Figure 6.8 CPN for the penetration stage of the FTP Bounce intrusion (ftp-bounce-cpn) . 103
Figure 6.9 CPN for the entire NFS Filehandle Guessing intrusion (nfs-filehandle-hier-cpn) 105
Figure 6.10 CPN for the reconnaissance stage of the NFS Filehandle Guessing intrusion (nfs-filehandle-scan-cpn) ......................................................... 106
Figure 6.11 CPN for scanning for a host providing the NFS service (scan-nfs-cpn) ........ 106
Figure 6.12 CPN for scanning for a host providing the Rlogin service (scan-rlogin-cpn) ... 107
Figure 6.13 CPN for the penetration stage of the NFS Filehandle Guessing intrusion (nfs-filehandle-cpn) ................................................................. 107
Figure 6.14 Rules for Sendmail anomaly detection ......................................................... 109
Figure 6.15 RIPPER rules for Sendmail modeled as a CPN .......................................... 109
Figure 6.16 Sample Java interface for a Place .............................................................. 111
Figure 6.17 Sample Java interface for a Token Matcher ................................................. 112
Figure 6.18 Sample Java interface for a Transition ....................................................... 112
Figure 7.1 IDS development process ............................................................................. 117
Figure 7.2 Unconstrained AND node with corresponding CPN ................................. 126
Figure 7.3 AND node, constrained by “Y after X”, with corresponding CPN ............... 128
Figure 7.4 AND node, constrained by “Y immediately after X”, with corresponding CPN. 128
Figure 7.5 OR node with corresponding CPN ............................................................. 129
Figure 7.6 Fault tree for FTP bounce attack, with constraints ...................................... 131
Figure 7.7 Template CPN from SFT for FTP bounce attack ....................................... 132
Figure 7.8 Template CPN for the FTP bounce attack after augmentation with source places 133
Figure 7.9 Fully augmented CPN for the FTP bounce attack ...................................... 134
Figure 7.10 General CPN for IDS ............................................................................. 135
Figure 7.11 Sample DCPN with duplicated leaves ...................................................... 136
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1.1</td>
<td>Network intrusions classified using proposed model attributes</td>
<td>28</td>
</tr>
<tr>
<td>Table 1.2</td>
<td>UNIX intrusions classified using proposed model attributes</td>
<td>29</td>
</tr>
<tr>
<td>Table 1.3</td>
<td>UNIX network service intrusions classified using proposed model attributes</td>
<td>30</td>
</tr>
<tr>
<td>Table 1.4</td>
<td>Windows intrusions classified using proposed model attributes</td>
<td>31</td>
</tr>
<tr>
<td>Table 1.5</td>
<td>Windows network service intrusions classified using proposed model attributes</td>
<td>31</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Agents and their related intrusions</td>
<td>50</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Sample system call windows with training labels</td>
<td>56</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Sample system call windows with classifications</td>
<td>56</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Results of learning rules for feature vectors</td>
<td>59</td>
</tr>
<tr>
<td>Table 4.4</td>
<td>Feature subset selection results with constant genetic parameters</td>
<td>63</td>
</tr>
<tr>
<td>Table 4.5</td>
<td>Results from rules learned by genetic feature selection</td>
<td>63</td>
</tr>
<tr>
<td>Table 4.6</td>
<td>Effectiveness of various learning techniques</td>
<td>64</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I gratefully acknowledge the support of my wife, Holly Helmer, in my pursuit of higher education and for her constant support of my graduate studies. Together we have been through much over the past four years, and I look forward to many more years together as we raise our family and continue our careers.

I thank Dr. Johnny Wong who has encouraged and assisted me in many ways. My understanding of distributed systems, computer security, and intrusion detection have been greatly expanded under his guidance as my major professor and instructor in many courses. I appreciate all of the time and effort he has expended on my behalf. It's been enjoyable and useful as well to work with many of Dr. Wong's master's students on intrusion detection and computer security projects.

Dr. Vasant Honavar has taught me so much about artificial intelligence, machine learning, and agent systems. I appreciate the countless hours he has spent assisting with my research in machine learning and genetic feature selection for intrusion detection problems, agent systems for intrusion detection, and building models for intrusions and intrusion detection.

Dr. Les Miller has provided guidance and input throughout my program. His probing questions about aspects of the models developed for intrusions and intrusion detection have strengthened the models and their analysis. He and his students have also worked to apply data warehousing to the intrusion detection problem, and I appreciate all of their work.

Dr. Doug Jacobson and Dr. Jim Davis have been great resources for computer security and TCP IP network implementation discussions. The "Information Warfare" courses offered by Drs. Jacobson and Davis were interesting, informative, and fun to take.

Thank you to Dr. Robyn Lutz for constructive input into this dissertation as I have explored models for intrusions and intrusion detection. I also appreciate the time I spent working with her on software safety issues at the requirements specification level for product families.

I have learned the usefulness of discourse from all my committee members. Through presentations and discussions, my committee has helped direct my research, focus my interests, and extend the results
far beyond anything I could have hoped to accomplish on my own.

Several master’s students in computer science have assisted and supported my research activities. In particular, I appreciate Mark Slagell’s help with aspects of the mobile agent intrusion detection system.

Dr. Arthur Oldehoeft provided outstanding support for my pursuit of graduate studies. He took time from his busy schedule to meet my wife and me when we first visited Iowa State University’s Department of Computer Science in January of 1994. Dr. Oldehoeft and the entire departmental faculty and staff have been extremely supportive and helpful, and they have made it possible for me to complete my graduate work.

Dr. John Gustafson, formerly of Ames Laboratory, renewed my research interest in high-performance computing. Under his guidance, I learned about the current state of the art in high performance computing, developing high-performance programs for scientific computation on parallel systems, and analyzing performance of parallel systems.

Thanks also to others at at the Ames Laboratory Scalable Computing Lab, including David Halstead, Don Heller, Ricky Kendall, Rajat Todi, Brett Bode, Vasily Lewis, and Brian Smith.

I am grateful for the use of the Design CPX colored petri net tool software courtesy of the CPX group at the University of Aarhus, Denmark [28]. I am also grateful for the use of the SAPHIRE fault tree tool software courtesy of the Idaho National Engineering and Environmental Laboratory and the Nuclear Regulatory Commission [51].

Assistance by Johnny Wong, Vasant Honavar, and Les Miller is gratefully acknowledged for help with chapter 4. Assistance from Johnny Wong, Robyn Lutz, Mark Slagell, Vasant Honavar, and Les Miller is gratefully acknowledged for help with chapters 5, 6 and 7.

The majority of this work was funded by the U.S. Department of Defense. A portion of this work was performed at Ames Laboratory under Contract No. W-7405-Eng-82 with the U. S. Department of Energy. The United States government has assigned the DOE Report Number IS-T 2065 to this thesis. A portion of this work was performed with support from Rockwell Collins, Inc. Research Grant 155-735. A portion of this work was performed with support from the Boeing Company in the form of the Boeing Dissertation Fellowship. A portion of this work was performed with support from the Graduate College of Iowa State University.
Intelligent mobile agent systems offer a new approach to implementing intrusion detection systems (IDS). The prototype intrusion detection system, MAIDS, demonstrates the benefits of an agent-based IDS, including distributing the computational effort, reducing the amount of information sent over the network, platform independence, asynchronous operation, and modularity offering ease of updates. Anomaly detection agents use machine learning techniques to detect intrusions; one such agent processes streams of system calls from privileged processes. Misuse detection agents match known problems and correlate events to detect intrusions. Agents report intrusions to other agents and to the system administrator through the graphical user interface (GUI).

A sound basis has been created for the intrusion detection system. Intrusions have been modeled using the Software Fault Tree Analysis (SFTA) technique; when augmented with constraint nodes describing trust, contextual, and temporal relationships, the SFTA forms a basis for stating the requirements of the intrusion detection system. Colored Petri Nets (CPN) have been created to model the design of the Intrusion Detection System. Algorithmic transformations are used to create CPN templates from augmented SFT and to create implementation templates from CPNs. The implementation maintains the CPN semantics in the distributed agent-based intrusion detection system.
1 INTRODUCTION

This chapter begins with a brief overview of computer security, security policy, and misuse & anomaly intrusion detection. Then, sources of audit information and organization of attacks into temporal stages are introduced to form a basis for later chapters that discuss key components of the distributed agent-based intrusion detection system. The chapter concludes with a brief glossary of terms and the thesis statement.

1.1 Computer Security and Intrusion Detection

A secure computer system provides guarantees regarding the confidentiality, integrity, and availability of its objects (such as data, processes, or services). However, systems generally contain design and implementation flaws that result in security vulnerabilities. An intrusion takes place when an attacker or group of attackers exploit security vulnerabilities and thus violate the confidentiality, integrity, or availability guarantees of a system. Intrusion detection systems detect some set of intrusions and execute some predetermined action when an intrusion is detected. Intrusion detection systems may also detect attempts at intrusions and use the information gained to determine appropriate action when an intrusion takes place.

1.1.1 Security Policy

A security policy defines the standards of integrity, availability, and exclusivity at some level in a computing system. Enforcement of the security policy is accomplished (insofar as is possible) by the hardware and software of the computing system. An intrusion is a violation of that security policy.

However, a formal security policy for a distributed system may not be well-defined and may possibly depend on system security policies. Multi-user operating systems and network protocols implement some default security policy. Administrators of the systems can adjust the security settings and install software that enforces further policy components (e.g., tcp wrappers [131]). Some attempts at violating a system's security policy may be flagged by the system (e.g., multiple failed logins). Successful violations
of the security policy may be difficult to detect and require an intrusion detection system to collect & correlate data and detect the violation (e.g., GrIDS [123], MuSigs [73]) or to detect evidence of the violation (e.g., Tripwire [58]).

1.1.2 Misuse Detection

A misuse intrusion detection system searches for occurrences of events which have been previously identified as intrusions. Intrusion detection systems which make use of misuse detection features include IDES, NIDES, & EMERALD (using the P-BEST expert rule system [75]), GrIDS [123], MuSigs [73], USTAT [52], and many current commercial and freely-available systems (including Network Flight Recorder, NetRanger, and SNORT).

Misuse intrusion detection systems detect matching intrusions with reasonably good accuracy (although such systems may be fooled by crafty attackers [109]).

Misuse-based intrusion detection systems have several disadvantages. Only known intrusions can be identified, so newly discovered vulnerabilities can be exploited without being detected. Signatures may be difficult to develop for certain intrusions. A misuse IDS may flag intrusions that were merely unsuccessful attempts because the particular vulnerabilities are not present in the monitored system. Expert systems are only as good as the encapsulated knowledge and also may require maintenance over time.

1.1.3 Algorithms for Misuse Detection

1.1.3.1 Pattern Matching

Many misuse intrusions may be detected by patterns that match some audit data. Kumar [62] uses computational complexity to classify patterns in a hierarchy:

1. **Existence** - Matching evidence left behind by an intruder, such as changed file permissions. Existence checking tends to have constant computational complexity.

2. **Sequence patterns** - Matching events in order, which is potentially an NP Hard computational complexity problem.

3. **Regular Expression patterns** - Conjunctions of sequence patterns, which is superset of sequence patterns and thus is potentially an NP Hard problem as well.

4. **Other patterns** - All other intrusion signatures.
Example misuse detection systems based on pattern matching include:

- Colored Petri Net Patterns from Purdue University [63]
- MuSigs from George Mason University [73]
- GrIDS from University of California, Davis [123]
- Snort by Martin Roesch of Stanford Telecommunications, Inc. [113]

### 1.1.3.2 Expert Systems

Knowledge is obtained from computer security experts and encoded into a rule set. One of the best-known expert systems used for misuse intrusion detection is P-BEST, which was used in the MIDAS, IDES, NIDES, and EMERALD intrusion detection systems [75][103].

### 1.1.4 Anomaly Detection

Anomalous events may be detected by finding significant deviations from predicted or expected behavior profiles. Assuming normal activity of a computing system falls well within the security policy, then anomalous activity is a likely indicator of an intrusion. Numerous algorithms have been developed to detect anomalies, including the NIDES statistical component [3], Computer Immunology [40][132], our own work with anomaly detection using feature vectors [48][49][47], and Lane & Brodley’s instance-based learning [65].

Successfully using anomaly detection in intrusion detection systems has been a goal of intrusion detection for at least the past thirteen years [31]. Anomaly-based intrusion detection has a number of advantages over misuse intrusion detection. It may not be necessary to have examples of misbehavior to develop a profile of expected or predicted behavior. An anomaly detection system does not depend on prior knowledge of possible misuses. An anomaly detection system can help discover new patterns of misuse.

Unfortunately, anomaly detection presents a set of challenges [65]. The temporal window of interest may vary from seconds to months. The volume of event data that must be analyzed may be huge (e.g., system calls executed by processes). The representation of events affects the interpretation and long-term storage of the data. Normal usage patterns may change over time. Innocuous anomalies (false positives) may occur frequently, thus obscuring true intrusions (true positives) and reducing user confidence. Profiles developed from “normal” data containing intrusions results in a flawed profile. Profiles that are adapted to changes in normal behavior over time may be “trained” by attackers to
accept intrusive behavior. If a threshold is used to discriminate between normal and anomalous behavior, it may be difficult to balance false positives versus false negatives (the receiver operating characteristic (ROC) convex hull method may be used to find a sound choice for such a threshold [108]).

1.1.5 Algorithms for Anomaly Detection

1.1.5.1 Specification-Based Profiles

Sekar et al. [121] have developed an anomaly detection system which profiles transitions in state machines. The intrusion detection system flags events which fail to match the previously-learned profiles.

1.1.5.2 Learning Algorithms

A wide variety of machine learning algorithms may be used for anomaly detection. Such algorithms used for anomaly detection include:

- Instance-Based Learning from Purdue University [65]
- RIPPER from AT&T Research [26], used by JAM at Columbia [67] and Warrender at New Mexico [132] for anomaly detection

1.1.5.3 Statistical Profiles

An anomaly intrusion detection system can learn a statistical profile for events based on intensity measures, relative distribution measures, categorical distribution measures, and ordinal measures. Implementations of statistical profiles for anomaly detection include:

- IDES from SRI International [31]
- XIDES from SRI International [3]
- EMERALD from SRI International [103]

1.1.5.4 Immunological Approach

Applying the highly effective concepts of animal immunology to anomaly detection and countermeasures is an intriguing idea. Immune systems learn to distinguish normal from abnormal and attack abnormalities when encountered. Algorithms for an immunological approach to anomaly detection have been developed in these projects:
1.2 Audit Records

Denning [31] defined the attributes of audit records in terms of subjects and objects. In this dissertation, the characteristic attributes of audit trail data are expanded to include network events.

1.2.1 Audit Data Attributes

- Event Source, which may include one or more of
  - User¹
  - IP Address
  - TCP Port
  - UDP Port
  - Hostname
  - Program & Process ID

- Event Target, which may include one or more of
  - User
  - Process ID
  - IP Address
  - TCP Port
  - UDP Port
  - Hostname
  - Filename

- Event Count

¹Source IP address, source hostname, and source user may be spoofed or forged under certain protocols. This information may be unreliable, but under normal operation these attributes are reliable and useful for development of profiles for anomaly detection. An example of unreliable data is the TCP SYN spoofing attack, which uses TCP packets created with a forged source IP address. The issue of reliability of these attributes could be addressed by assigning a reliability estimate to the values, and reducing the estimated reliability if an event is correlated with another intrusive event.
• Starting Time

• Duration or Ending Time

• Action

• Result (success, failure, error code)

• Source of Event Data (sensor identification)

1.2.2 Data Sources in Systems and Networks

Operating systems and networks are designed using layered models. Each layer in a model provides specific services and interfaces to higher layers. Layers separate a problem into pieces to reduce the overall complexity and to hide implementation details [127][128]. Instrumentation at the various layers of a system provide different types and amounts of information that has different uses. For example, system calls (at the kernel layer) executed by processes provide a large quantity of data useful for anomaly detection [47][49][132][67][41]. User logins and logouts would be best obtained from a system's authentication mechanism (which, in the case of UNIX, is managed by the application layer) and can be used for anomaly or misuse detection.

Layers in host systems which may be instrumented include:

• Applications

• Daemons    System services

• File system

• Memory management

• Process management

• Devices

Network protocol layers which may be instrumented include:

• Applications

• Transport

---

2Time synchronization between systems is necessary to enable fusion of event records based on timestamps. Unfortunately, time synchronization itself may be a target of attacks. An attacker may forge network time messages, resulting in incorrect adjustments of system clocks.
As an illustration of needing to obtain data from appropriate layers of systems, consider Network Intrusion Detection (NID) systems. NID systems often eavesdrop on individual IP packets at the Network layer. At this layer, it is simple to find Network-layer attacks such as Smurf [23] or Teardrop [21]. However, discovery of intrusions at the higher network layers requires the NID to re-assemble fragments, interpret packet contents, and resequence out-of-order segments to construct an internal model of the activity at the Transport and Application layers. The internal model may be inaccurate due to latent faults in the actual implementations of the Transport and Application layers on hosts and routers, differences between the traffic seen by the NID and the traffic that actually reaches the target, or ambiguities in protocol specifications. Placek and Newsham examine several such problems with NID systems and illustrate resulting vulnerabilities of NID systems themselves [109]. Data collected directly from the involved systems at the appropriate layers would avoid such problems.

### 1.2.3 Means of Obtaining Audit Data

Systems and networks are not normally designed to provide audit data to an intrusion detection system. Systems typically report abnormal events via a system logging facility and report most user authentication events, but extensive logging at all levels of a system is not normally performed. Network layers are not instrumented in any standard fashion, with the exception of audit trails available from firewalls. Thus, data must be obtained from a variety of other sources, including system logs, add-on programs, network sniffers, and network management protocols.

#### 1.2.3.1 System Log Files

Log files contain information relating to all layers of a host system. Files containing logged information must be parsed and interpreted to obtain audit records.

Applications, daemons, and kernel layers typically use the system logging facility (syslogd) on Unix systems. Each log message has an associated facility (including kernel, user, mail, daemon, auth, syslog, lpr, news, uucp, cron, authpriv, ftp, ntp, and security) and priority (including emergency, alert, critical, error, warning, notice, info, and debug). Log messages may be included or excluded from log files based on the facility and priority. Also, log messages may be sent
as UDP datagrams to syslog daemons on other systems. Figure 1.1 shows sample log file entries from applications (su), daemons (sendmail, ntpd, lpd, and cron), and kernel layers.

System log messages may easily be spoofed by users, either on a host system via the syslog daemon's named pipe (usually /dev/log or /var/run/log) or on the network via UDP datagrams to port 514.

A few binary files also contain auditing information on Unix systems. The wtmp file (usually found in /etc, /var/adm, or /var/log) contains fixed-length records for each user login and logout event. System shutdown, startup, and time change events are also recorded in the wtmp file. The utmp file (usually found in /etc, /var/adm, or /var/run) contains fixed-length records describing current user sessions. When processes terminate, process accounting information may be written to acct or pacct (usually found in /var/adm or /var/account).

```
Feb 28 09:52:22 hostname sendmail[66144]: JAA66144: from=user, size=1264, class=0, pri=31264, nrcpts=1, msgid=<200002281552.JAA66144Qhostname.cs.iastate.edu>, relay=user®localhost
Feb 28 09:52:35 hostname sendmail[66146]: JAA66144: to=user@cs.iastate.edu, ctladdr=user (1060/1060), delay=00:00:13, xdelay=00:00:13, mailer=esmtp, relay=css-l.cs.iastate.edu. [129.186.3.24], stat=Sent (JAA01230 Message accepted for delivery)
Feb 28 12:40:32 hostname ntpd[143]: using kernel phase-lock loop 2041
Feb 28 12:40:33 hostname lpd[185]: restarted
Feb 28 12:46:26 hostname su: user to root on /dev/tty5
Feb 28 12:50:32 hostname /kernel: xl0: promiscuous mode enabled
Feb 28 13:00:00 hostname CRON[480]: (root) CMD (newsyslog)
```

Figure 1.1 Sample syslog entries

1.2.3.2 Process Traces

The system calls executed by processes may be obtained from most systems. On Unix systems, the tracing program may be named ktrace, strace, or truss. Systems that perform C2-level auditing may provide audit data similar to system call traces. Each program outputs data in a different format, so the output must be parsed and interpreted appropriately.

1.2.3.3 Host Status Monitoring

The operational status of a host can be a useful indicator for intrusion detection. Characteristics that may be monitored include network interface packet statistics, system uptime, system task load.

---

3 It may be infeasible to trace all processes due to processing and disk storage overhead.
disk I/O rate, number of users, and amount of unallocated disk space. Unix programs that can be used to monitor these aspects of a system include `netstat`, `uptime`, `iostat`, and `df`.\(^4\)

### 1.2.3.4 Eavesdropping

Eavesdropping may be performed on network packets on broadcast networks or point-to-point links.\(^5\) This is the primary technique used to obtain network auditing data by most network intrusion detection systems. An issue with eavesdropping intrusion detection systems is where the sensor(s) should be placed. Sensors can be positioned between outside networks and a site's firewall, between a firewall and a site's internal network, or even between internal networks [100]. Multiple sensors that can correlate events between themselves may help detect intrusions that would otherwise go unnoticed, such as internal attacks or evasive attacks [109].

Eavesdropping may also be used to obtain information from user login sessions (via "tty snooping").

### 1.2.3.5 SNMP Queries

The Simple Network Management Protocol (SNMP) [11] provides a common protocol for management of systems that implement the IP protocol. Managed systems can include network infrastructure devices such as routers, switches, hubs, and modems. Multi-user systems, including Unix and Windows workstations and servers, can also be managed via SNMP if the service is installed and enabled.

SNMP defines a language (a subset of ASN.1 syntax) that identifies managed objects in a system. A Management Information Base (MIB) [83] for each monitored system defines the available attributes and describes each attribute. A manager can query attributes of the managed system via SNMP and receive responses which describe the state of the managed system. As with host status monitoring, SNMP queries can be used to collect status data for an intrusion detection system. Information available from a monitored system often includes uptime, network interface statistics, and CPU load.

Unfortunately, SNMP is not a secure protocol. SNMP requests are authenticated by a plain text password (or "community", in SNMP parlance). Many SNMP implementations use the default password "PUBLIC", and some devices allow attributes to be changed via the default password. SNMP messages may be spoofed with trivial effort.

---

\(^4\)Output from standard system programs may not be trustworthy if the system has been compromised by an intruder who has installed replacement programs which hide his activity (typically as part of a rootkit).

\(^5\)As with process tracing, it may be infeasible capture all data from eavesdropping due to the volume of the data and the overhead involved in processing all the data.
1.2.3.6 SNMP Traps

SNMP traps [11][114] can be sent asynchronously by systems to a manager. The traps that are available depend on the managed system. Many systems can issue traps for network problems and for security-related incidents. An intrusion detection system can make use of such traps for intrusion detection. However, SNMP traps can be forged and must be treated as unreliable data.

1.2.4 Relationship of Available Data Sources to Graph Model

Sensors at different points and levels of a distributed system will see parts of an intrusion. For example, in the FTP Bounce Attack intrusion of Figure 1.2, imagine a firewall with a network sensor between the Attacker Host and other two hosts. This network sensor will identify the FTP network activity between the Attacker Host and FTP Host but will not see the result of the attacker’s GET command (the connection between the FTP Host and the Target Host). Thus another sensor, placed strategically so that activity between the internal hosts can be eavesdropped, would be required to fill in the missing information.

![Diagram of an FTP Bounce Attack](image)

Figure 1.2 Diagram of an FTP Bounce Attack
1.3 Intrusion Taxonomy

An analysis of intrusions is offered to enable fusion of intrusion information from multiple data sources in the multi-agent intrusion detection system. A taxonomy is proposed, and various known intrusions are examined under the developed taxonomy. An intrusion taxonomy is defined by Amoroso as "a structured representation of intrusion types that provides insight into their respective relationships and differences" [2]. The aspects of intrusion classification are drawn from analysis of hundreds of known intrusions gathered by Guy Helmer [46], the Bugtraq mail list [120], CERT Advisories [24], CIAC Advisories [27], various vendors, and numerous other sources. Previous work in the area, including [98] and [74], were also consulted. Based on the previous classification work, dimensions identified for the taxonomy include temporal & spatial aspects, techniques & target components, outcomes, and potential responses. Attributes of the dimensions are further investigated below.

Temporal Aspects Temporally, intrusions tend to progress over time in stages ranging from reconnaissance to penetration and entrenchment. Identifying the stage of the attack leads to appropriate countermeasures, e.g. countermeasures may be softer in response to reconnaissance than to penetration. Amoroso outlines the simple temporal model of intrusions but simply classifies actions as either benign or malicious [2, Chapter 4].

Attacks tend to progress in temporal stages. Ruij identifies seven temporal stages in successful attacks: reconnaissance, vulnerability identification, penetration, control, embedding, data extraction & modification, and attack relay [115].

Examples of reconnaissance and vulnerability identification include DNS table gathering, host and port scanning, host operating system identification, and password sniffing.

Penetration includes exploitation of various network server daemon vulnerabilities (poor authentication and buffer overflows), authenticating with illicitly obtained passwords, and TCP session hijacking.

Control includes installing Trojan, backdoor, and other rootkit programs, removing traces of the intrusion from system logs, and disabling detection systems.

Embedding involves the installation or modification of a system so that even if the attacker is discovered and steps are taken to recover the system, the attacker will still be able to enter the system. For example, the system bootstrap code could be modified to re-insert backdoors if the system executable programs are restored from backups or installation media.

In the data extraction and modification phase, the attacker gathers information about the configuration and operation of the system. Covert channels are suggested as a good way to move discovered
data from the compromised system to the attacker's base. An example of useful extracted data would be cracked account passwords.

After a system is fully compromised, it may be used for attack relaying. Attacks can be launched against affiliated (trusting) hosts to expand the number of hosts under the attacker's control. A system also may simply be used to participate in distributed denial of service attacks [35][33][34][32].

Spatial Aspects We have identified attacks that include spatial aspects. An attack may be distributed across several hosts in a subnet or several departments of an organization. Implicit or explicit trust relationships (often insufficiently authenticated) between components of a distributed system in an organization allow an attack's effects to propagate beyond a single component.

Attack Target The target of an attack affects how, where, and at what level the attack can be detected. Neumann and Parker developed the following taxonomy [98]. which generally ranges from physical hardware through operating systems and system management. The Neumann and Parker taxonomy is NP1-External Misuse, NP2-Hardware Misuse, NP3-Masquerading, NP4-Subsequent Misuse, NP5-Control Bypass, NP6-Active Resource Misuse, NP7-Passive Resource Misuse, NP8-Misuse via Inaction, and NP9-Indirect Aid. We further identify the layer of the host system (device, device driver, kernel, file system, memory manager, process manager, daemon, application) or network protocol (physical, data link, network, transport, application) targeted by each attack.

Outcome and Lethality The outcome and lethality of an attack affects the priority and required response to the attack. The four levels of outcomes are normal, suspicious, attempted (failed) attack, and successful attack. Outcomes of successful attacks include unauthorized disclosure of data or use of services (violating exclusivity), unauthorized modification of data (violating integrity), and denial of service (violating availability). Lethality ranges from low (typical of reconnaissance attacks) to high (typical of penetration attacks).

Responses Responses to intrusions may include documenting the attack, automatically intervening in the progress of the intrusion, remediating the changes due to the intrusion, deception or misdirection, redirection & containment, and prevention of future intrusions.
1.4 Descriptions of Classified Intrusions

The following descriptions of intrusions are grouped by attacks on the lower layers of the network protocol stack (OSI layers 3 & 4), host-based attacks, and attacks on network applications (OSI layer 7).

1.4.1 Network Protocol Intrusions

The following intrusions are oriented to flaws in networking protocols or flaws in software that implements the networking protocols. This section concentrates on OSI layers 3 (network) and 4 (transport).

1.4.1.1 Host Scanning

Attackers may use one of these techniques to identify hosts for further inspection.

**PING Scanning**  Attackers may iterate through the possible Internet Protocol (IP) addresses for systems and send an ICMP ECHO REQUEST to each address. Systems which respond with an ICMP ECHO reply may be targeted for port scanning.

**Broadcast PING**  The broadcast PING (an ICMP ECHO REQUEST to a broadcast destination IP address) has two uses for attackers. The broadcast PING can be used to identify all hosts on the target’s subnet. In this case, the source IP address will be valid so that the attacker can receive the ICMP ECHO replies.

**Smurf**  The broadcast PING can also be used to create a denial of service (known as the “smurf” attack) by amplifying single, frequent requests which generate many more ICMP ECHO responses that will clog a network link or slow a target host. In this case, the source IP address will often be forged to be the target of the denial of service.

1.4.1.2 Port Scanning

Attackers use port scanning to identify TCP and UDP services that are offered by a particular host. Note that attackers may attempt to hide the true source IP address of the attack with decoys (simultaneously generating identical scanning packets with other source IP addresses). **nmap** is a powerful tool that implements all of the following scanning methods and more.
**TCP SYN Port Scanning** An attacker sends TCP segments with the SYN flag set (the standard first segment in TCP connection establishment) to determine which ports are listening on the target host. Target ports which are open for listening will respond with a SYN|ACK packet. This method works through firewalls when connections are allowed to the port on the target host, but fails when a firewall does not allow connections to the port on the target host.

**TCP Stealth Port Scanning** An attacker sends forged TCP segments with the FIN flag set to determine which ports are listening on a host. Target ports which have a server listening will not respond; otherwise, the target host usually responds with an RST (reset). This scan method works through many packet filtering firewalls, since packets without the SYN flag set are often forwarded without further checking.

Variations on this scanning method use different flag values, including no flags (null), ACK, ACK|PSH|FIN, or ACK|FIN.

**UDP Port Scanning** Because UDP ports are connectionless, UDP ports can (so far as is known) only be scanned by sending a UDP packet to a particular port and then observing whether an ICMP port unreachable error response packet is received. This technique fails if ICMP packets are blocked between the destination and source. This technique also fails if a firewall does not allow packets to the port on the target host.

### 1.4.1.3 Finger

The Finger protocol [136] provides either a list of active users or information about a specific user. If the finger service is active on a host, an attacker can learn:

- Whether a system administrator is currently logged on
- What user names are available on a system
- Personal information about a user that may assist password guessing

### 1.4.1.4 Reverse DNS Spoofing

An attacker with authority over a reverse lookup domain name service table (which defines the mapping from IP addresses to domain names for a network) can spoof the domain name for his or her computer [119]. The attacker can:
1. Change the table to return the name of a host that is trusted by the attacker's target

2. Connect to the target using rsh or rlogin

3. If the target accepts the spoofed reverse domain name and trusts the computer by that name, the attacker will not be required to give a password

### 1.4.1.5 DNS Cache Poisoning

This attack is similar to the reverse DNS spoofing attack (section 1.4.1.4). An attacker that can observe or receive DNS requests from the target's domain name server can spoof a response that includes not only the requested data but also unassociated, poisoned data that the target's DNS server will add to its cache [19]. The attacker can:

1. Poison the target's name server cache with a reverse DNS record that gives the attacker's computer the name of a host trusted by the target
2. Connect to the target using rsh or rlogin
3. If the target accepts the poison domain name and trusts the computer by that name, the attacker will not be required to give a password

### 1.4.1.6 DNS Zone Transfer

An attacker looking for targets may be able to obtain the DNS zone file for the target organization or department. The DNS zone file lists each of the DNS names for hosts controlled by the organization and the associated IP addresses. The attacker can use this information to discover the layout of the target's network and the hosts within the target's domain.

The attack procedure is as easy as:

1. Starting nslookup
2. Using `set type=ns` to get name server records
3. Entering the domain name of the target domain (e.g., `ee.iastate.edu`) to obtain the name servers for the domain
4. Using `server v.x.y.z` to set the target name server to one of the IP addresses reported by the previous step
5. Using `ls target.domain` to list the contents of the target domain's zone file
1.4.1.7 FTP Bounce

The FTP bounce attack can be used to transfer data to a port to which an attacker does not
normally have access [20]. One way to exploit this problem is to use it to send data to a remote shell
server that trusts the FTP server. After an attacker discovers an FTP server and a host running rsh
that might trust the FTP server, the attacker:

1. Uploads a specially-formatted file to the FTP server

2. Issues an FTP PORT command that directs the FTP server to send its next download to port
   514 on the target host

3. Issues an FTP GET command to “download” the contents of the previously-uploaded file into
   port 514 on the target

The GET command opens a connection from the FTP server on port 20 to the rsh daemon on the
target. If the target trusts the FTP server, the rsh daemon will accept the contents of the file as if it
were user input and execute the given command.

1.4.1.8 TCP SYN Spoofing Denial of Service

The exploitation of the TCP SYN flood vulnerability takes advantage of the Transmission Control
Protocol (TCP) connection establishment procedure, usually called the "three-way handshake" [16][38].
The three-way handshake works as shown in figure 1.3.

Figure 1.3 TCP connection establishment diagram
When a TCP client on host A wants to connect to a TCP server on host B:

1. Host A sends a SYN (synchronization) segment to Host B, which includes A's initial sequence number $x$.

2. Host B responds with a SYN/ACK (synchronization acknowledgment) segment. This packet acknowledges Host A's SYN with sequence number $x + 1$, and announces its initial sequence number $y$.

3. Host A acknowledges B's sequence number by responding with $y + 1$. This fully establishes the connection.

After B receives the initial SYN from A, it must store information about the "embryonic" connection in an in-memory data structure. The operating system has a limited number of entries for embryonic connections. When no more entries are available, B will be unable to accept any more connection requests until some of the embryonic connections have been fully established or have timed out.

In a SYN flood attack, an attacker sends a high volume of SYN packets to the target without responding to the target's SYN/ACK responses. When the target's limited number of table entries are consumed, the target can no longer accept TCP connection requests. After the table fills, the attacker can continue to send SYN packets to keep the table full as previous entries slowly expire. Note that because the attacker does not need to receive the target's SYN/ACK packets, the source IP address can be forged to hide the identity of the attacker's host.

### 1.4.1.9 Daemon Buffer Overflows

Many daemons that provide network service run as a privileged user. Daemons that read or copy user input into a fixed size buffer may be vulnerable to buffer overflow vulnerabilities if the data is not consistently checked against the length of the buffer [72]. Because buffers are often stored as a local variable on the stack, user data that exceeds the length of the buffer can overwrite the procedure's return address. Data containing object code and a return address can be sent to the daemon by a malicious user. If the buffer overflow is successful, the malicious code is executed with privilege. The malicious code can make illicit changes to files on the system or open network connections to the attacker's computer.

This technique is the most useful and actively exploited method of gaining privileged access to network-connected systems. Buffer overflow attacks are difficult to detect and defend.
1.4.1.10 TCP Session Hijacking

If the segments in an active TCP connection can be observed, it is possible for the attacker to insert data into the TCP stream with valid acknowledgment numbers [61]. This can be used to insert commands into a telnet or rlogin session. If a user is logged into an administrative account at the time, it would be possible for an attacker to insert commands to modify the system configuration that would then allow the attacker direct access to the system.

The legitimate user of the TCP session will likely notice that the connection is disrupted. The TCP acknowledgment numbers will be unsynchronized between the server and the legitimate user’s computer. It is likely that no further legitimate communication will be possible over the connection.

1.4.1.11 Forged Invalid IP Fragment (Teardrop)

The teardrop attack sends forged IP fragments that overlap [21]. When a destination computer receives IP fragments, it must reassemble the fragments to construct the complete packet. Some IP stacks (including older Linux kernels and unpatched Windows NT and Windows 95 systems) do not correctly handle overlapping fragments and crash, resulting in a denial of service.

Related variations include bonk, boink, jolt, and newtear. An attack tool by the name of targa implements all of these variations [95].

1.4.1.12 Long ICMP ECHO REQUEST (Ping of Death)

An ICMP ECHO REQUEST (ping) with a payload size of 65510 crashes certain IP stacks, resulting in a denial of service [14]. The 65510-byte payload with an 8-byte ICMP header and 20-byte IP header results in an illegal 65538-byte IP packet.

1.4.1.13 Forged Invalid TCP SYN (Land)

A TCP SYN (connection establishment) packet is forged with the target’s IP address specified as both the source and destination and an open server port is specified as both the source and destination port. Several TCP stacks will either crash or enter an infinite loop when such a packet is received, resulting in a denial of service [21].

1.4.1.14 Out of Band TCP Data

The Windows NT TCP stack crashes when it receives a TCP segment with the URG flag set but the urgent offset pointer points to the end of the segment and no normal (non-urgent) data follows [86].
1.4.1.15 Simple UDP Services Bounce

Several simple UDP services reply with a UDP packet when they receive a UDP packet. Such services include echo, chargen, daytime, and time. An attacker can forge a UDP packet with any of the simple UDP services specified for both the source and destination port. A storm of UDP packets between the source and destination IP address will result, resulting in a denial of service due to network congestion [17]. Alternately, if the source and destination IP address are the same, the target computer’s kernel will spend most of its time processing the packets, resulting in a reduction of service.

1.4.1.16 NFS File Handle Guessing

Given an NFS file server, it may be possible to guess file handles for files exported by the file server [130]. Leendert van Doorn’s nfsbug.c program:

1. Makes an RPC connection to the mountd service on an NFS file server
2. Makes an RPC connection to the file service on an NFS file server
3. Checks for file systems that are exported to the world
4. Tries guessing the file handle of a filesystem’s root inode (inode = 2) based on typical device major & minor numbers, using the fact that the root inode of a file system often has an easy-to-guess generation number (handles are computed as a function of device major, device minor, inode number, generation number)

If the root inode of a file system can be found, all files and directories below that point can probably be accessed.

1.4.1.17 RSH SYN Spoofing

SYN spoofing allows an attacker to masquerade his computer as a trusted computer [7]. Assume an attacker learns that host A trusts rsh connections from host B. Attacker:

1. Uses a denial of service attack against host B to keep it busy or kill it (so B will be unable to quickly send an RST after it sees A’s forthcoming SYN|ACK)
2. Opens one or more valid TCP connections with A to determine the state of A’s Initial Sequence Number generator
3. Spoofs a TCP SYN with B's IP address as the source, A's IP address as the destination, and port 514 (shell) as the destination port. This causes A to believe that B is creating a TCP connection to its rsh daemon

4. Spoofs a TCP ACK with the expected ISN so that A believes it has a fully open TCP connection to its rsh daemon from B

5. Spoofs a TCP segment to A's rsh daemon that changes a configuration file

1.4.2 Host Intrusions

Intrusions which attack a host operating system are discussed in this section.

1.4.2.1 Password Guessing

Authenticated network services offer several ways to guess passwords for accounts. Passwords are often easy to guess. Some systems contain default passwords for system accounts. Many users typically use poor passwords that are based on names or dictionary words.

There are the obvious methods of using telnet or ftp to connect to a system and try passwords. However, systems will usually log multiple authentication failures that occur through telnet or ftp, leaving a trace that can lead back to the intruder. Other authenticated network services, such as the Post Office Protocol (POP), usually do not log multiple authentication failures nor do they close the connection after some number of failures. Automated password guessing programs exist that connect to the POP service and try passwords for given users until a password succeeds or the dictionary of passwords is exhausted.

If a list of encrypted passwords can be obtained from the target site, attackers can discover the plaintext for the passwords by using programs such as crack (for UNIX passwords) or L0phtcrack (for UNIX and Windows passwords).

1.4.2.2 Signals to Setuid Programs

A user may use signal handling vulnerabilities in privileged programs to gain privileges. An example is a vulnerability in sendmail version 8.7 [15]. This version of sendmail would execute the program "smtpd" when it received a hangup signal (SIGHUP). An attacker could

1. Create a symbolic link named "smtpd" to /bin/sh

2. Execute sendmail
3. While sendmail is waiting for input, use "kill -1 pid". where pid is the process ID of the sendmail process.

Now the attacker will have a copy of the shell /bin/sh executing with root privileges.

1.4.2.3 Temporary File Races

Temporary files are usually created in a directory which is publicly writeable, such as /tmp. Temporary files often have guessable names, based on the process ID of the creating process. If an attacker knows that a privileged process is about to create or open a temporary file for writing, the attacker can create a symbolic link in the public directory with the same name as the process's temporary file. If the symbolic link points to a critical file, such as /etc/passwd, and if the attacker has guessed the name of the temporary file correctly, the process will open the file to which the symbolic link points.

If the attacker is able to force information to be written to a system configuration file, he may be able to increase his privileges on the system. A privileged program may grant privileges to a user if particular files exist or contain specific information. For example, the remote shell daemon, rshd, will grant login rights to any user without requiring a password if /etc/hosts.equiv contains the line "+ +".

The privileged process may try to reduce the chance of this attack by issuing an access(2) system call to check for the existence of a file or symbolic link before creating the temporary file. This merely reduces the window of opportunity for the attacker, since the symbolic link could still be created between the time the access(2) call is made and the time the open(2) system call is executed.

1.4.2.4 Reading or Writing Critical Memory

Disclosure and compromise of critical system data presents serious problems. If an attacker can read arbitrary data from the system's memory space, the attacker may gain access to privileged information. Unencrypted passwords, authentication keys, and other private information may be stored in memory and may be vulnerable.

If an attacker can write to arbitrary addresses in the system's memory space, all protections implemented by the kernel can be compromised. For example, an attacker could modify his credentials and become the system administrator.

UNIX Memory Device Special Files In UNIX, the device special files /dev/kmem and /dev/mem provide user-level access to the kernel memory space and the physical memory space of the system. re-
spectively. Access to system memory via these device files is provided so that system utilities can show system status, including information about the process table, network protocol statistics, and virtual memory statistics. Access to these files must be strictly limited. Programs that execute with privileges to access system memory often include ps, vmstat, w, and netstat. Vulnerabilities in these programs or unrestricted access to these special files can allow attackers to gain privileges.

**Windows NT SecHole** An intrusion utility named Sechole.exe changes the object code executed by the OpenProcess API so that a subsequent request for debug rights allows the process to obtain elevated privileges [91].

### 1.4.2.5 Trojan Programs

An attacker can install modified versions of programs that, when executed by another user, obtain information for the attacker or modify permissions such that the attacker gains new privileges. An example of this attack could be carried out like this:

1. The attacking user on a system creates a modified version of the ls program in his or her home directory. The Trojan ls lists files just like the original, but also copies /bin/sh to /tmp/sh and makes it setuid to the root user.

2. The user complains to the system administrator that something is wrong with the files in his or her account.

3. The system administrator changes his or her current working directory to the user's home directory and executes the ls program to see what files are in the user's directory. If the system administrator's path includes '.', the current directory, the user's Trojan ls is executed.

4. The attacking user executes the setuid copy of the shell created by the Trojan ls to obtain increased privileges.

### 1.4.2.6 Rootkits

A "rootkit" is a package of programs that are typically installed by an attacker after he or she has gained root privileges on a system. The programs in the rootkit hide the attacker's activities and allow "backdoor" access to the system. Programs included in a rootkit typically include:

- ps
Newer rootkits replace many access control, password database management, and system utilities [122]. A Windows NT and Windows 2000 rootkit, ntrootkitOSl, has recently appeared but has very limited capabilities.

1.4.2.7 Backdoor Access

An attacker that has gained access to a system may install a “backdoor” program that will allow privileged access to the system in the future. This may be useful for the attacker if the system administrator closes the original hole that was used to compromise the system or if the original exploit required a large investment of time or work. The attacker may also use the backdoor to provide easy access for other intruders. Typical backdoors include:

- A modified login program allows a particular user to login without a password and gives root privileges to that user

- A network daemon that accepts telnet connections and runs a shell with root privileges (like rshd, but without any authentication checks)

The Linux Root Kit version 4 [122] includes several standard programs with backdoors, including chfn, inetd, login, and rshd.

1.4.2.8 Trim or Remove System Logs

System audit logs provide traces of an attacker’s activity. This information can be used to trace the attacker and may provide legal evidence that could be used against the attacker in court. Tools such as wipe, utclean, and remove exist to remove traces of the attacker’s activities from the login file (/var/log/wtmp), list of currently logged-in users (/var/run/utmp), and last login times (/var/log/lastlog) [122]. Attackers may simply erase all logs.
1.4.2.9 Disabling Intrusion Detection Systems

An intrusion detection system which is implemented as one or more processes running on a system can be killed by a root-privileged intruder. An intrusion detection system which is implemented as a passive network monitor can be overwhelmed by high traffic loads.

1.4.2.10 Password Sniffing

An intruder may install a password sniffer to obtain passwords for accounts on other systems. Telnet, FTP, POP, and other authenticated network protocols usually transmit passwords as cleartext. Many systems offer a facility, such as the Berkeley Packet Filter, which can be used to hear all traffic on the network to which the system is attached. Password sniffing programs use this eavesdropping facility to listen to the first few bytes of each network connection. The overheard information is saved to a file or transmitted to another host for storage. An intruder can use the passwords to gain access to accounts on other systems. The Linux Root Kit version 4 includes a password sniffer [122].

1.4.3 Network Application Intrusions

This section examines known intrusions that target OSI layer 7 (applications).

1.4.3.1 Sendmail Debugging Commands

The sendmail mail transfer daemon, prior to version 5.59 [12], contained two commands for remote debugging, `wiz` and `debug`. These two commands could be issued remotely to a sendmail daemon to allow execution of arbitrary commands as the root user. After the Morris worm incident highlighted this vulnerability, most publicly-accessible systems should have been upgraded to eliminate this problem.

1.4.3.2 RPC Portmapper Proxy

An attacker can use the `PMAPPROC_CALLIT` remote RPC (procedure number 5, the protocol forwarding service) indirectly to invoke other RPC functions, such as the NFS `mountd RPCMNT_MOUNT` function. For example, if a computer exports file systems to itself via NFS, and if the `mountd` daemon accepts requests from port numbers greater than 1023, and if the `portmap` daemon executes `pmap_rmtcall` functions for untrusted clients, this problem can be exploited to obtain NFS file handles (see also section 1.4.1.16).

Reference: bugtraq thread of August 1994 entitled “RPC protocol problem?” beginning with message ID `<199408230539.AA13138@ph-meter.beckman.uiuc.edu>`.
1.4.3.3 Unauthorized Use of NFS

Most implementations of NFS use the RPC \texttt{AUTH_UNIX} authentication mechanism, which authenticates requests only on the source IP address and source port number. If attackers obtain a valid NFS file handle, NFS requests can be spoofed with trusted IP source addresses. The spoofed requests can create or modify files on the target server [130].

1.4.3.4 Trivial File Transfer Protocol

The Trivial File Transfer Protocol (tftp) is often used to provide software updates or boot kernels to simple network devices, including routers, X terminals, and diskless workstations. tftp does not use any authentication. Any file available via tftp can be retrieved by anyone who can send requests to the tftp port, UDP port 69 [13]. If the files available via tftp include password files or other security configuration information, this information can assist an attacker. If the tftp daemon allows files to be written, anyone who can send requests to the tftp port can upload files. If the files writable by tftp include password files or other security information, the system can be easily compromised by an intruder.

Modern tftp daemons implement strict limitations on file access, but can still be configured in an insecure manner.

1.4.3.5 Use of rlogin or rsh From Untrusted Hosts

Other attacks, including those mentioned in sections 1.4.1.4, 1.4.1.5, 1.4.1.7, and 1.4.1.17, setup attacks on rlogin or rshd. Both protocols authenticate based on the source IP address of the TCP connection and the DNS name obtained via reverse lookup. If an attacker successfully spoofs the IP address or name of a trusted host, the attacker is given access to a shell prompt [16].

1.4.3.6 Insecure Web Server CGI Programs

The NCSA and Apache httpd web server daemons were installed with sample Common Gateway Interface (CGI) programs that contained vulnerabilities that allowed attackers to execute arbitrary commands on the server. Vulnerable CGI programs include phf [18] and nph-test-cgi [22].

An attacker can build an HTTP request like "http://server.net/cgi-bin/phf?Qalias\%0acat\%20/etc/passwd" that will exploit the vulnerability in the phf program and cause it to return the contents of the system's password file. An attacker can then use a password cracker (mentioned in section 1.4.2.1) to obtain plaintext passwords for further access to the system.
1.4.3.7 Microsoft Internet Information Server (IIS)

Various versions of IIS contain security holes which allow access to private information, creation of files, or cause the server to crash.

Showcode IIS Version 4.0 contains an Active Server Page named showcode.asp which allows any remote user to view any text file on the web server [64]. As an example, an HTTP URL request could be written like this:


which would obtain the contents of WS_FTP's WS_FTP.ini file, which contains encoded passwords for FTP sites visited by the computer's user. The encoded passwords may be decoded by the decoder at http://www.hispasec.com/wsftp.asp.

MS-DOS Style Names IIS 3.0 and 4.0 may allow attackers to access protected files with alternate ("8.3"-style) names [89][90]. As an example, say an attacker sends a request to a vulnerable server for a known file that contains sensitive information, such as salary information: the URL

http://server.net/protected-dir/salaries.htm

fails with a permission denied error, while the URL

http://server.net/protected/salaries.htm

succeeds.

Alternate Data Streams IIS may allow attackers to access protected files using alternate names for a file's DATA stream [93][92][87]. NTFS associates multiple streams with each file. A user may specify the default stream. DATA. by appending the string "::$DATA" to the pathname to a file. IIS normally executes Active Server Pages and sends the results to the user as HTML, but a user may obtain the source code for the Active Server Page (which may contain privileged information) by appending "::$DATA" to the URL.

1.4.3.8 NT NULL Session

Windows NT allows anonymous connections (known as NULL session connections) through which account names and names of shared resources are visible [88]. Attackers may use this vulnerability to
obtain lists of user names, network file shares, and other resource information. This information gives an attacker information that can be used to assist penetration of the target.

1.5 Classifications using Intrusion Taxonomy

Tables 1.1, 1.2, 1.3, 1.4, and 1.5 show classifications of various intrusions using the proposed model attributes.

A problem that is evident in the examination of intrusions under the proposed taxonomy is the level of abstraction applied to the intrusions. Identification and classification of vulnerabilities [82] can give clues to appropriate level of abstraction, since intrusions bear a close relationship to vulnerabilities. However, vulnerability analysis tends to focus more on the causes rather than the identification of intrusions [8]. Abstraction in the taxonomy examined here ranges from specific attacks (e.g., most network protocol intrusions) to very general (e.g., buffer overflow attacks). This variation in abstraction is due to the dissimilarities in attack spatiality, target, outcome, and lethality.

1.6 Glossary

Audit Record A unit of information that uniquely identifies an action taking place in a computing system, including time of action, location where action took place, initiator of the action, and object acted upon.

Anomaly Action that deviates from normal activity.

Attacker An individual who attempts one or more attacks against a system.

Attack An action by an individual that violates or attempts to violate a security policy.

Backdoor A method of access to a system that bypasses normal authentication.

CPN Colored Petri Net.

False Negative Notation for an intrusive event that was incorrectly identified by an intrusion detection system as normal.

False Positive Notation for a normal event that was incorrectly identified by an intrusion detection system as intrusive.

Intrusion A collection of attacks by an attacker or group of related attackers to achieve a single goal.
Table 1.1  Network intrusions classified using proposed model attributes

<table>
<thead>
<tr>
<th>Intrusion</th>
<th>Temporal</th>
<th>Spatial</th>
<th>Target</th>
<th>Outcome</th>
<th>Lethality</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNS Zone Transfer</td>
<td>Recon</td>
<td>Multiple DNS Servers</td>
<td>NP7 Network: DNS Application</td>
<td>Disclosure of Data</td>
<td>Low</td>
<td>Deception</td>
</tr>
<tr>
<td>Broadcast PING</td>
<td>Recon</td>
<td>All Hosts</td>
<td>NP7 Network: ICMP Protocol</td>
<td>Disclosure of Data</td>
<td>Low</td>
<td>Deception</td>
</tr>
<tr>
<td>(Smurf)</td>
<td>Operation</td>
<td>All Hosts</td>
<td>NP7 Network: ICMP Protocol</td>
<td>Denial of Service</td>
<td>High</td>
<td>Prevention</td>
</tr>
<tr>
<td>Host Scan</td>
<td>Recon</td>
<td>All Hosts</td>
<td>NP7 Network: ICMP, TCP, &amp; UDP Protocols</td>
<td>Disclosure of Data</td>
<td>Low</td>
<td>Prevention, Deception</td>
</tr>
<tr>
<td>Port Scan</td>
<td>Recon</td>
<td>Single Host</td>
<td>NP7 Network: TCP &amp; UDP Protocols</td>
<td>Disclosure of Data</td>
<td>Low</td>
<td>Prevention, Deception</td>
</tr>
<tr>
<td>Xmas</td>
<td>Recon</td>
<td>Single Host</td>
<td>NP7 Network: TCP Protocol</td>
<td>Disclosure of Data</td>
<td>Low</td>
<td>Deception</td>
</tr>
<tr>
<td>Finger</td>
<td>Recon</td>
<td>Single Host</td>
<td>NP7 Network: TCP Protocol</td>
<td>Disclosure of Data</td>
<td>Low</td>
<td>Deception</td>
</tr>
<tr>
<td>Reverse DNS</td>
<td>Penetration</td>
<td>Single Host</td>
<td>NP3 Network: Insecure TCP App</td>
<td>Use of Service</td>
<td>High</td>
<td>Terminate connection</td>
</tr>
<tr>
<td>Spoofing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DNS Cache Poisoning</td>
<td>Penetration</td>
<td>Single Host</td>
<td>NP3 Network: Insecure DNS Caching Server</td>
<td>Modification of Data</td>
<td>Moderate</td>
<td>Remove incorrect data</td>
</tr>
<tr>
<td>FTP Bounce</td>
<td>Penetration</td>
<td>One or Two Hosts</td>
<td>NP5 Network: Insecure TCP App</td>
<td>Use of Service</td>
<td>High</td>
<td>Terminate connection</td>
</tr>
<tr>
<td>TCP SYN Spoofing</td>
<td>Penetration</td>
<td>Single Host</td>
<td>NP5 Network: Insecure TCP App</td>
<td>Use of Service</td>
<td>High</td>
<td>Terminate connection</td>
</tr>
<tr>
<td>Daemon Buffer Overflows</td>
<td>Penetration</td>
<td>Single Host</td>
<td>NP5 Network: Insecure TCP or UDP App</td>
<td>Use of Service</td>
<td>High</td>
<td>Terminate connection</td>
</tr>
<tr>
<td>TCP Session Hijacking</td>
<td>Penetration</td>
<td>Single Host</td>
<td>NP5 Network: TCP Protocol</td>
<td>Use of Service</td>
<td>High</td>
<td>Terminate connection</td>
</tr>
<tr>
<td>Password Guessing</td>
<td>Penetration</td>
<td>All Hosts</td>
<td>NP5 Host: User Authentication</td>
<td>Use of Service</td>
<td>High</td>
<td>Lock target account</td>
</tr>
<tr>
<td>Forged Invalid IP Fragment (Teardrop)</td>
<td>Operation</td>
<td>Single Host</td>
<td>NP6 Network: IP fragment reassembly</td>
<td>Denial of Service</td>
<td>Moderate</td>
<td>Restart target host</td>
</tr>
<tr>
<td>Forged Invalid TCP SYN (Land)</td>
<td>Operation</td>
<td>Single Host</td>
<td>NP6 Network: TCP connection establishment</td>
<td>Denial of Service</td>
<td>Moderate</td>
<td>Restart target host</td>
</tr>
<tr>
<td>Simple UDP</td>
<td>Operation</td>
<td>Multiple Hosts or Network</td>
<td>NP6 Network: UDP packet bouncing</td>
<td>Denial of Service</td>
<td>Moderate</td>
<td>Terminate inetd</td>
</tr>
<tr>
<td>Intrusion</td>
<td>Temporal Spatial</td>
<td>Spatial Target</td>
<td>Outcome</td>
<td>Lethality</td>
<td>Response</td>
<td></td>
</tr>
<tr>
<td>-----------------------------------------------</td>
<td>------------------</td>
<td>----------------</td>
<td>----------------------</td>
<td>-----------</td>
<td>------------------</td>
<td></td>
</tr>
<tr>
<td>Signals to setuid programs</td>
<td>Pen.</td>
<td>Single Host</td>
<td>NP5 Host: Tamper with running process</td>
<td>Use of Service</td>
<td>High Disable user</td>
<td></td>
</tr>
<tr>
<td>Privileged Program Overflows</td>
<td>Pen.</td>
<td>Single Host</td>
<td>NP5 Host: Insecure privileged program</td>
<td>Use of Service</td>
<td>High Terminate process Disable user</td>
<td></td>
</tr>
<tr>
<td>Exploit temporary file race to create or overwrite files</td>
<td>Pen.</td>
<td>Single Host</td>
<td>NP6 Host: Change file system</td>
<td>Modification High of Data</td>
<td>Revert changes</td>
<td></td>
</tr>
<tr>
<td>Write to protected memory</td>
<td>Pen.</td>
<td>Single Host</td>
<td>NP6 Host: Change kernel memory</td>
<td>Modification High of Data</td>
<td>Restart system</td>
<td></td>
</tr>
<tr>
<td>Install Trojan Programs</td>
<td>Pen.</td>
<td>Single Host</td>
<td>NP4 Host: Change file system</td>
<td>Modification Moderate of Data</td>
<td>Revert changes</td>
<td></td>
</tr>
<tr>
<td>Execute Trojan Program</td>
<td>Pen.</td>
<td>Single Host</td>
<td>NP6 Host: Grant privileges</td>
<td>Modification High of Data</td>
<td>Revert changes</td>
<td></td>
</tr>
<tr>
<td>Execute Trojan Program</td>
<td>Pen.</td>
<td>All Hosts</td>
<td>NP6 Host: Change file system</td>
<td>Modification High of Data</td>
<td>Revert changes</td>
<td></td>
</tr>
<tr>
<td>Install &quot;rootkit&quot; Camo.</td>
<td>Camo.</td>
<td>Single Host</td>
<td>NP6 Host: Change file system</td>
<td>Modification High of Data</td>
<td>Revert changes</td>
<td></td>
</tr>
<tr>
<td>Install Backdoors Entrench</td>
<td>Camo.</td>
<td>Single Host</td>
<td>NP4 Host: Change file system</td>
<td>Modification Moderate of Data</td>
<td>Revert change</td>
<td></td>
</tr>
<tr>
<td>Use Backdoor for Access</td>
<td>Operation</td>
<td>Single Host</td>
<td>NP5 Host: Use of Service</td>
<td>Moderate</td>
<td>Terminate sessions</td>
<td></td>
</tr>
<tr>
<td>Remove System Logs Camo.</td>
<td>Camo.</td>
<td>Single Host</td>
<td>NP6 Host: Change file system</td>
<td>Modification Moderate of Data</td>
<td>Alert security officer</td>
<td></td>
</tr>
<tr>
<td>Disable Detection Systems Entrench</td>
<td>Entrench</td>
<td>Single Host</td>
<td>NP6 Host: Tamper with processes</td>
<td>Modification High of Data</td>
<td>Restart detection systems</td>
<td></td>
</tr>
<tr>
<td>Install Password Sniffer Operation</td>
<td>Single Host</td>
<td>XP7 Network: TCP Protocols &quot;telnet&quot;, &quot;ftp&quot;</td>
<td>Disclosure High of Data</td>
<td>High</td>
<td>Terminate process</td>
<td></td>
</tr>
<tr>
<td>Password Guessing Pen.</td>
<td>Pen.</td>
<td>Multiple Hosts</td>
<td>NP3 Host: Masquerade as another user</td>
<td>Use of Service</td>
<td>Moderate Temporarily block accounts</td>
<td></td>
</tr>
</tbody>
</table>
Table 1.3 UNIX network service intrusions classified using proposed model attributes

<table>
<thead>
<tr>
<th>Intrusion</th>
<th>Temporal</th>
<th>Spatial</th>
<th>Target</th>
<th>Outcome</th>
<th>Lethality</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sendmail</td>
<td>Pen.</td>
<td>Single</td>
<td>NP5 Host: Use of authentication</td>
<td>Use of</td>
<td>High</td>
<td>Deception: reduced level of service to source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Host</td>
<td></td>
<td>Service</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RPC portmapper proxy</td>
<td>Pen.</td>
<td>Single</td>
<td>NP5 Host: Insufficient authentication</td>
<td>Use of</td>
<td>High</td>
<td>Blacklist or reduce level of service to source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Host</td>
<td></td>
<td>Service</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NFS</td>
<td>Pen.</td>
<td>Single</td>
<td>NP6.NP7 Host: Disclosure or Modification of Data</td>
<td>Disclosure</td>
<td>High</td>
<td>Blacklist or reduce level of service to source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Host</td>
<td></td>
<td>or Modification of Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tftp</td>
<td>Pen.</td>
<td>Single</td>
<td>NP8 Host: Unprotected service</td>
<td>Disclosure</td>
<td>High</td>
<td>Blacklist or reduce level of service to source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Host</td>
<td></td>
<td>or Modification of Data</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rlogin, rsh from untrusted host</td>
<td>Pen.</td>
<td>Single</td>
<td>NP5 Host: Improper modified configuration, DNS information attacked</td>
<td>Use of</td>
<td>High</td>
<td>Terminate session: Blacklist source</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Host</td>
<td></td>
<td>Service</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 1.4 Windows intrusions classified using proposed model attributes

<table>
<thead>
<tr>
<th>Intrusion</th>
<th>Temporal</th>
<th>Spatial</th>
<th>Target</th>
<th>Outcome</th>
<th>Lethality</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Privileged Program Buffer Overflows</td>
<td>Pen.</td>
<td>Single</td>
<td>NP5 Host: Insecure privileged program</td>
<td>Use of High</td>
<td>Service</td>
<td>Terminate process disable user</td>
</tr>
<tr>
<td>Write to protected memory</td>
<td>Pen.</td>
<td>Single</td>
<td>NP6 Host: Change kernel memory</td>
<td>Modification High of Data</td>
<td>Revert</td>
<td>Restart system</td>
</tr>
<tr>
<td>Install Trojan Programs</td>
<td>Pen.</td>
<td>Single</td>
<td>NP4 Host: Change file system</td>
<td>Modification Moderate of Data</td>
<td>Revert</td>
<td>change</td>
</tr>
<tr>
<td>Execute Trojan Program</td>
<td>Pen.</td>
<td>All Hosts</td>
<td>NP6 Host: Grant privileges</td>
<td>Modification High of Data</td>
<td>Revert</td>
<td>change</td>
</tr>
<tr>
<td>Execute Trojan Program</td>
<td>Pen.</td>
<td>All Hosts</td>
<td>NP6 Host: Change file under user's control</td>
<td>Modification High of Data</td>
<td>Revert</td>
<td>change</td>
</tr>
<tr>
<td>Install Backdoors</td>
<td>Entrench</td>
<td>Single</td>
<td>NP4 Host: Change file system</td>
<td>Modification Moderate of Data</td>
<td>Revert</td>
<td>change</td>
</tr>
<tr>
<td>Use Backdoor for Access</td>
<td>Operation</td>
<td>Single</td>
<td>NP5 Host: Avoid authentication</td>
<td>Use of Moderate Service</td>
<td>Terminate improperly authenticated sessions</td>
<td></td>
</tr>
<tr>
<td>Remove System Logs</td>
<td>Camo.</td>
<td>Single</td>
<td>NP6 Host: Change file system</td>
<td>Modification Moderate of Data</td>
<td>Revert</td>
<td>change</td>
</tr>
<tr>
<td>Disable Detection Systems</td>
<td>Entrench</td>
<td>Single</td>
<td>NP6 Host: Tamper with running processes</td>
<td>Modification High of Data</td>
<td>Restart</td>
<td>detection systems</td>
</tr>
<tr>
<td>Password Guessing</td>
<td>Pen.</td>
<td>Multiple</td>
<td>NP3 Host: Masquerade as another user</td>
<td>Use of Moderate Service</td>
<td>Temporarily block targeted accounts</td>
<td></td>
</tr>
</tbody>
</table>

### Table 1.5 Windows network service intrusions classified using proposed model attributes

<table>
<thead>
<tr>
<th>Intrusion</th>
<th>Temporal</th>
<th>Spatial</th>
<th>Target</th>
<th>Outcome</th>
<th>Lethality</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIS Attacks</td>
<td>Pen.</td>
<td>Single</td>
<td>NP5 Host: Avoid authentication</td>
<td>Use of High</td>
<td>Service</td>
<td></td>
</tr>
<tr>
<td>IIS Index Server</td>
<td>Operation</td>
<td>Single</td>
<td>NP7 Host: Access to Files</td>
<td>Disclosure</td>
<td>Moderate</td>
<td></td>
</tr>
<tr>
<td>NULL Session Account Name Recon</td>
<td>Recon</td>
<td>Single</td>
<td>NP7 Host: Access to account information</td>
<td>Disclosure</td>
<td>Moderate</td>
<td></td>
</tr>
</tbody>
</table>
Misuse Action known to violate security policy.

Rootkit A collection of programs an attacker may install in the embedding stage of an intrusion. The programs may:

- hide the attacker’s activity, including network connections and processes:
- allow the intruder backdoor access to the system:
- collect confidential data from processes, files, or network interfaces.

See also section 1.4.2.6.

Security Policy Definition of allowed activities by users on objects (including files, devices, and other users).

True Negative Notation for a normal event that was correctly identified as non-intrusive by an intrusion detection system.

True Positive Notation for an actual intrusive event that was correctly identified as intrusive by an intrusion detection system.

Vulnerability A weakness in a system that allows actions contrary to the security policy.

1.7 Thesis Statement and Outline

Is it possible to build a distributed agent-based system for intrusion detection based on sound engineering techniques for misuse detection and machine learning techniques for anomaly detection?

Prior work in intrusion detection has examined misuse and anomaly detection in both monolithic and distributed systems. This work applies mobile agent technology to provide a distributed intrusion detection system operating within a distributed computing system. A portion of the work targeted the addition of intelligent anomaly detection agents into the system. Other parts of the work develop engineering-based requirements and design specifications for the distributed agent-based system based on temporal aspects of intrusions.

Chapter 2 reviews related intrusion detection systems and algorithms and discusses differences from our work.

Chapter 3 introduces the intrusion detection system based on mobile agents. An overview of agent systems is given, and arguments for using mobile agents for intrusion detection are presented.
Chapter 4 shows the use of a feature vector representation to describe the system calls executed by privileged processes. A rule learning algorithm is used to induce rules that can be used to monitor the system for anomalies. The performance of the rule learning algorithm on this task with and without feature subset selection using a genetic algorithm is examined. Feature subset selection is shown to significantly reduce the number of features used while improving the accuracy of predictions.

Chapter 5 examines the use of Software Fault Tree Analysis to model intrusions and develop requirements specifications for the intrusion detection system.

Chapter 6 introduces Colored Petri Nets (CPNs) for intrusion detection systems. CPNs are a well-documented and well-used abstraction used for modeling complex and distributed systems. CPNs provide a convenient model for describing the gathering, classification, and correlation activities of an intrusion detection system. CPNs appear to provide a clean design upon which to base further development of our agent-based IDS.

Chapter 7 integrates the Software Fault Tree-based requirements, Colored Petri Net-based design, and mobile agent implementation of an intrusion detection system.

Chapter 8 offers conclusions and ideas for future work.
2 RELATED WORK

This chapter describes intrusion detection systems and algorithms that are related to the mobile agents intrusion detection system. None of the related intrusion detection systems use a distributed Colored Petri Net, combine anomaly detection with misuse detection, or develop a requirements model for intrusion detection.

2.1 Intrusion Detection System Architectures

Related system architectures for intrusion detection systems include CIDF and DIDS.

2.1.1 Common Intrusion Detection Framework

The Common Intrusion Detection Framework (CIDF) [112] resembles our project’s architecture. CIDF is a standard proposed by a group including the Information Technology Office of the Defense Advanced Research Projects Agency, University of California-Davis, Information Sciences Institute, Odyssey Research, and others.

At the bottom layer, the CIDF reconnaissance agents correspond to our data gathering agents. In the middle, the CIDF analysis agents correspond to our low-level agents. At the top layer, the CIDF decision-response agents correspond to our high-level agents. The similarity of our prototype system to the CIDF model and the flexibility of the Common Intrusion Specification Language (CISL) [37] should lend itself to integration with other CIDF-compatible systems when the need arises.

2.1.2 Distributed Intrusion Detection System

The Distributed Intrusion Detection System (DIDS) of the University of California-Davis [96] uses a combination of host monitors and local area network monitors to monitor system and network activities. A centralized director aggregates information from the monitors to detect intrusions.

DIDS is similar to our agent system for intrusion detection and countermeasures in that it uses multiple monitors and artificial intelligence algorithms to determine the severity of events. DIDS differs
from our system in that the intelligence is purely centralized, and DIDS does not make use of any agent technology.

2.2 Graph-based Models

Related graph-based modeling techniques and implementations of intrusion detection systems, including GrIDS, MuSigs, IDIOT, and Specification-based ID, are briefly reviewed.

2.2.1 Graph-Based Intrusion Detection System

The Graph-Based Intrusion Detection System (GrIDS) developed at the University of California-Davis dynamically builds graphs describing network activity by applying user-defined rules to audit data [123]. In GrIDS graphs, nodes represent hosts or aggregations of hosts and edges represent network activity. Rather than building a single graph including all system activity, individual graphs are maintained by rule sets. Each rule set matches certain events from the network audit trail and either builds a new graph or adjusts an existing graph. Rule sets may also combine graphs and raise alerts when suspicious patterns are discovered.

The motivating example given by the GrIDS designers is the graphical model of a worm such as the Morris Worm [36]. As the worm spreads through systems, a tree-shaped graph would be developed by GrIDS as in Figure 2.1.

![Figure 2.1 Beginning of a worm graph, and the graph after the worm has spread [123]](image)

In the GrIDS approach, nodes model hosts and edges model activities. The model allows intuitive aggregation of nodes and edges into reduced graphs. Reduced graphs allow higher level analysis and data sharing, resulting in a scalable design.

GrIDS detects violations of security policy, and as such is inherently a misuse detection system.
However, it should be possible to extend the graphical model to select objects and events to analyze for anomalies.

Our specification and design is not limited to modeling communication patterns in networks but includes the entire distributed system. Rather than directly using the graphical model, a mobile agent intrusion detection system is developed and implemented using graph-based models as the requirements and design specification.

### 2.2.2 Misuse Signatures

The Adaptable Real-time Misuse Detection system (ARMD) developed at George Mason University represents misuses as directed acyclic graphs (DAGs) [73]. Abstract events are represented by nodes in a graph. Edges in a graph represent the ordering of inter-event rules satisfied by the nodes. Nodes are selected for the graph by intra-event rules. The inter- and intra-event rules together define misuse signatures (named MuSigs). If a graph is built such that a sink node has an edge to it, an intrusion is detected.

Unlike the object event model used by GrIDS, MuSig graphs are not amenable to aggregation. Edges in a MuSig graph only mean a predicate has been satisfied; edges then have no values or attributes that can be aggregated. Nodes in a MuSig graph correspond to specific events: aggregation of events seems to be a difficult proposition in the absence of structured methods to aggregate the attributes associated with the events. Finally, MuSig graphs can not be used for anomaly detection since by definition a MuSig graph detects a misuse intrusion. Thus, the GrIDS-style object event model seems more powerful for general misuse and anomaly intrusion modeling.

Our CPN models allow for aggregation through unification of tokens and also allows for anomaly detection. Our intrusion detection system is developed using the CPN for the design specification, so our IDS does not need to implement matching of graphs. Our SFTA approach models intrusions, not intrusion detection.

### 2.2.3 Intrusion Detection In Our Time

Kumar and Spafford developed a variant of CPNs for misuse detection [63] at Purdue, which implements pattern matching for intrusion detection. Intrusion Detection In Our Time (IDIOT). Kumar's intrusion detection system, implemented CPNs with modifications. Kumar eliminated concurrency, removed local condition variables at transitions, added start and final states, and added invariant conditions to patterns [62]. IDIOT used a custom language to describe patterns. To improve the
computational complexity of direct execution of CPNs. Various optimizations were used to reduce the computational effort involved in matching audit data to patterns.

Our IDS is developed using the CPN as a design specification rather than a direct execution of a CPN. We define a transformation from the CPNs to the implementation of the software agent intrusion detection system that preserves the CPN semantics. Our IDS also operates in a distributed environment using an agent-based approach. After modeling intrusions using SFTA, we use CPNs for modeling intrusion detection. Our CPNs are hierarchical and composable, and they model stages of attacks and are traceable to SFT-based requirements.

### 2.2.4 Specification-based Intrusion Detection

Sekir et al. at SUNY-Stonybrook have used state machine models of protocols for intrusion detection [121]. Their intrusion detection system detects anomalies when transition frequencies in a state machine exceed normal (learned) frequencies. Rather than modeling intrusive activity, their technique models all possible behaviors of a system. Their technique is oriented to anomaly detection.

Our CPN-based IDS handles both anomaly and misuse intrusions. It is not limited to a particular algorithm, so it could make use of the state-transition approach to anomaly detection as well as other anomaly detection algorithms.

### 2.3 Agent-based Intrusion Detection

Projects closely related to our intrusion detection project include JAM, AAFID, and IDAS.

#### 2.3.1 Java Agents for Meta-Learning

The Java Agents for Meta-Learning (JAM) project at Columbia University [124] is the most similar to our agent system for intrusion detection and countermeasures. JAM uses intelligent, distributed Java agents to learn models of fraud and intrusive behavior. The data mining aspects of JAM are discussed separately in section 2.4.1.

Our project differs from JAM in our use of mobile agents for intrusion detection. Our project also is looking at countermeasures that could be used to combat intrusions.

#### 2.3.2 Autonomous Agents for Intrusion Detection

The AAFID group at Purdue's CERIAS has prototyped an agent-based intrusion detection system. Their paper analyzes the agent-based approach to intrusion detection and mentions the prototype work
that has been done on AAFID [5].

Our project differs from AAFID in that we are using data mining to detect intrusions on multiple components, emphasizing the use of learning algorithms in intrusion detection, and using mobile agents. AAFID is implemented in Perl while our system is implemented in Java.

2.3.3 Intrusion Detection Agent System

The Intrusion Detection Agent system at the Information-Technology Promotion Agency in Japan watches for suspected intrusions, traces information related to the suspected intrusion, and determines whether an intrusion actually took place [4].

Our agent-based system shares data between agents to determine whether an intrusion has taken place. Our agents are continually collecting and monitoring information, rather than dispatching an investigator (tracing) agent after a suspicious event is noticed.

2.4 Machine Learning and Data Mining Approaches to Intrusion Detection

Machine learning and data mining for intrusion detection projects include JAM, Computer Immunology, and temporal sequence learning.

2.4.1 Java Agents for Meta-Learning

The Java Agents for Meta-Learning (JAM) project at Columbia University [124] uses intelligent, distributed Java agents and data mining to learn models of fraud and intrusive behavior. The JAM project expanded on the work done by Forrest’s group [41] to detect intrusions on privileged programs. Like the JAM project, we apply the data mining approach to intrusion detection but use a different data representation that provides a single signature for each process.

Our project differs from JAM in that we are concentrating on data mining within an organization. The JAM Project seemed to focus on data mining over multiple organizations.

2.4.2 Computer Immunology

The Computer Immunology project at the University of New Mexico [40] explored designs of intrusion detection systems based on animal immune systems. One portion of the project developed a sense of “self” for security-related computer programs by creating a database of normal system call traces from instances of execution of the programs [41]. This sense of self can be used to detect intrusions by comparing execution traces of processes to the database of system call traces. Work that is more
recent used a variety of approaches to detect intrusions using system call traces from several different privileged programs [132].

In general, the Computer Immunology project differs from our project by focusing on individual agents rather than sharing data between agents and feeding the data into a data warehouse. In our specific application of intrusion detection using system calls, we do not attempt to develop a sense of self but instead concentrate on developing rules to check a particular program’s execution trace for anomalies. We assume knowledge of the identity of the program being traced and apply a rule set learned for that particular program.

2.4.3 Temporal Sequence Learning

Terrance Lane at Purdue University developed a data mining algorithm for anomaly detection called Temporal Sequence Learning [65], an instance-based learning technique. Temporal Sequence Learning was applied to the shell commands executed by users to learn profiles that can distinguish between normal and anomalous sequences of commands.

Lane’s algorithm would be a candidate for inclusion in our agent-based system as another data mining agent. However, we have concentrated on rule-learning algorithms operating on feature vectors built from lists of system calls executed by privileged programs.

2.5 Summary

Related prior work performed on intrusion detection system architectures, models for intrusion detection, and agent-based IDSs was briefly examined and contrasted with our mobile agent intrusion detection system. Overviews were also given of influential machine learning and data mining projects dealing with anomaly intrusion detection.

After this point, the dissertation will deal with our work performed. First, the use of mobile agents in an IDS will be considered, including the use of dynamic aggregation for extending agent capabilities and communicating between agents. Then, an anomaly detection technique using a “bag of words”-inspired representation combined with the RIPPER rule learning algorithm will be detailed. Modeling intrusions using Software Fault Tree Analysis (SFTA) will be explored, and modeling intrusion detection using Colored Petri Nets (CPN) will also be examined. Finally, the connections between SFTA, CPN, and the agent implementation of the IDS will be shown.
3 MOBILE INTELLIGENT AGENTS FOR INTRUSION DETECTION

The multi-agents intrusion detection model is based on the idea of distributed knowledge networks. The model makes use of the following layers:

- User Interface and Feedback
- Data Warehousing
- Data Fusion and Aggregate Classification & Data Mining
- Data Gathering, Base Classification, and Basic Data Mining
- Data Cleaning and Formatting

This modular and extensible approach to building a system helps solve the complex problems in an intrusion detection system. It divides the problem into the problem of information retrieval, classification, collaboration, and compilation. Agents have been developed for our system that retrieve information from distributed systems, classify the data (either using embedded expert rules or machine learning techniques), and stores the data in a database. The fundamental design of the department-level intrusion detection system is illustrated in Figure 3.1.

At the bottom of the tiered architecture, the system log routers and system activity agents read log files and monitor the operation of the systems. The routers feed into the distributed data cleaning agents which previously registered their interest in particular events. Targeted data cleaning agents process data obtained from the routers and activity agents. They render the data into common data formats.

In the middle of the architecture, the low level, mobile agents form the first line of intrusion detection. They periodically travel to each of their associated data cleaning agents, obtain the recently-gleaned information, and classify the data to determine whether singular intrusions have occurred.

The low level agents are managed by mediators which control the systems visited by the agents, obtain the classified data from the agents, and route the data into the local database and to the user
Figure 3.1  Original architecture of the Mobile Agents Intrusion Detection System
interface. As the system is developed, the mediators will apply data mining algorithms to the data in
the database to connect individual events into a cohesive view of the elements involved in an attack.

Our implementation of the system currently includes:

- Static data gathering agents that mine information from system logs, audit data, and operational
  statistics and render the information in a common format:

- Low level agents that monitor and classify ongoing activities, classify events, and pass on their
  information to mediators:

- Facets for the low level agents that add cooperation to the agents:

- Data mining agents that use machine learning to acquire predictive rules for intrusion detection
  from system logs and audit data.

As we further develop the system, multiple departmental-level systems can be monitored. Data warehousing can be used to combine the knowledge and data from the individual departments into an organization-wide view of attacks. The agent system of Figure 3.1 is targeted to run at the departmental level of an organization. To provide enterprise-wide information about intrusions, the data from each departmental agent system will feed into a data warehouse as shown in Figure 3.2.

![Figure 3.2 Enterprise data warehouse for intrusion detection](image-url)
Because the data warehouse will provide a global view of the intrusion detection systems, it supports not only the identification of attacks but also:

- helps administrators discover new attacks.
- trains system administrators about how attacks are mounted on their systems, and
- identifies weak points in the enterprise information systems.

3.1 Agent Systems

Software agents consist of program code & state and exist to perform tasks on behalf of a user with some degree of autonomy. A software agent’s goal may require some degree of intelligence, allowing it to react to its environment, make plans to achieve its goal, maximize its utility, and or modify its behavior over time [50]. Software agents may use mobility to travel to sources of data and remotely execute their tasks, resulting in a natural distribution of work and reduced communication overhead.

A number of agent infrastructures exist to support agent systems. Generally, agents infrastructures provide agent servers, agent interfaces, and agent brokers. Agents servers may provide mobility and authentication. Agent interfaces are used by application programs to create and communicate with agents. Agent brokers provide naming and location services. The prototype mobile agent intrusion detection system has been built in the Java language using the Voyager Object Request Broker [101]. Voyager provides the mobility, interfacing, and naming services needed to implement the agents in the IDS. Research work is being performed on secured agent systems, such as the SMART agent system [133].

3.2 Motivation for Using Distributed Mobile Agents

Distributed mobile autonomous agents solve several critical problems in intrusion detection and provide a general architecture for adding and integrating “components” into the system. Monolithic, centralized systems have several faults which may be overcome by the use of a distributed architecture [53][54].

Network intrusion detectors typically use single sensors attached to network segments. However, local area networks have moved towards switched architectures which do not broadcast unicast frames to all network segments. Centralized sensors will miss traffic on segments to which the sensor is not
attached in a switched environment. Distributed agents solve this problem by monitoring network activity at each host.

Network intrusion detectors also have problems with high data rates. Centralized systems may miss packets under heavy load situations on Fast or Gigabit Ethernet networks. Distributed agents distribute the processing effort between the networked systems and likely would improve the chances that intrusions will be detected that would be missed by a centralized IDS.

The modular architecture of the agent-based intrusion detection system allows for the integration of intrusion detection components from other projects. For example, the SNORT network intrusion detection system [113] is a small, fast, low-overhead sensor that watches for network packets that match signatures. The agent-based intrusion detection system can incorporate SNORT and provide network misuse detection at each host. Likewise, various small programs detect specific intrusions such as port scanning, broadcast pings (smurfs), logins at unusual times, and changes to files. These can be included in the distributed intrusion detection system via wrapper agents.

A centralized or centrally-governed intrusion detection system provides a single point of failure and a single target for an attack. The distributed agent architecture avoids a central point of failure. Autonomous agents may continue operating despite the failure of other agent servers or other failures in a system, which avoids the compromise of the entire IDS even if one component fails or is attacked [84]. Mobile agents also may be capable of evading attackers and resurrecting themselves if killed.

A distributed agent-based intrusion detection system helps solve spatial problems in intrusion detection, where more than one host is involved in an intrusion. For example, in an "FTP bounce" attack, an attacker may use an anonymous FTP server on one host to spoof a command to a remote shell on the target host. Agents on both the anonymous FTP server and the target host could detect the spatially-separate events in the attack and correlate the events in near real-time.

Kumar [62] lists shortcomings of intrusion detection systems. Viewed in a different way, the shortcomings provide a list of desirable features in an intrusion detection system.

**Generic Architecture.** The Common Intrusion Detection Framework [102] (CIDF) specifies a generic architecture for an intrusion detection system and classifies the components of an intrusion detection system. A system of distributed mobile agents implements the intrusion detection system in a flexible way compatible with the CIDF architecture.

**Efficiency.** A distributed multi-agent system obtains audit data at the appropriate levels of the distributed system and distributes the information processing and intrusion detection effort.
Portability. Intrusion detection systems have tended to be developed with an orientation to an organization's security policy. Differences in security policies between organizations result in a lack of portability of an intrusion detection system. In a different sense, portability of the intrusion detection system with respect to operating systems and computer architecture is also an issue. Perl and Java, two interpreted languages, have been used to provide portability for intrusion detection systems. AAFID [5] and our own project, MAIDS, are two such IDSs.

Upgradability. An intrusion detection system based on a component-based architecture such as that available in an agent-based system satisfies the upgradability and enhancement concern. New features can easily be added to such a system.

Maintenance. Maintaining and updating the learned knowledge used by components of an intrusion detection system would depend on the architecture of the components.

Performance Benchmarks. Exhaustive quantitative performance evaluations of current intrusion detection systems in real-world environments do not exist. Vulnerability coverage is beginning to be addressed by vendors with the assistance of vulnerability assessment projects, including the Common Vulnerability Enumeration [82].

Testing. Kumar claimed, "there is no easy way to test intrusion detection systems." [62, page 35] The MIT Lincoln Labs intrusion detection evaluation [76] was one of the first major tests of research intrusion detection systems. Attacks remain difficult to simulate, effectiveness of systems is difficult to evaluate without operating them under real world loads, and significant tuning and expertise tends to be necessary to operate systems.

The multi-agent intrusion detection system is similar to other intrusion detection systems in that its effectiveness would need to be demonstrated in real-world settings.

3.3 Lightweight Agents

Agent systems with lightweight agent support allow runtime addition of new capabilities to agents. This section explains the addition of agent collaboration to the system by dynamically adding capabilities to lightweight agents.

We call the small, minimal agents lightweight because the agents implement a minimum of functionality, as opposed to heavyweight agents that include all functions that may ever be needed. Compared to heavyweight agents, lightweight agents are:
• Smaller:

• Simpler:

• Faster to transport (due to their smaller size):

• Dynamically updatable and upgradable.

Dynamic aggregation allowed us to add collaboration capabilities to the lightweight mobile agents and quickly add new features to the system. Our particular use of dynamic aggregation allows the agents in our intrusion detection system to inform each other about intrusive activity. Each agent uses dynamic aggregation to manage its sensitivity level. The sensitivity level determines how sensitive an agent is to events which may not be considered intrusive under normal circumstances but which can be intrusive in the presence of related intrusions. An example of the sensitivity issue is the problem of failed login attempts. A few login failures on a single host may be normal, such as when a user forgets her password. However, when an attacker has identified a target host (perhaps by portscanning, which is considered intrusive by our system), he may connect to the target host and try a few typical passwords. In this case, a loose temporal relation exists between the first event (portscanning) and the second event (failed login attempts). Agent sensitivity levels can provide a real-time correlation of related intrusions while allowing normal, individual events to pass without triggering alarms.

3.3.1 Adding Agent Capabilities with Dynamic Aggregation

Dynamic aggregation of objects offers three benefits [101].

• The behavior of a binary-only object may be extended.

• An object may be customized in specific ways.

• An object's behavior may be extended at runtime, in ways that need not be specified at compile time.

To this list of benefits we add:

• An object is as small as it can be until new functionality is needed.

The Voyager terminology for objects used in dynamic aggregation is “facets”. Voyager defines a primary object and its facets to be an “aggregate” that is managed as a single unit. Voyager’s facet selection
rules conveniently select a facet implementation tailored to a particular object. This allows specific customization of an object.

Voyager selects an implementation of a facet based on the name of the facet and the class name of the primary object. When a facet of a particular class is requested, Voyager searches for a facet that implements an interface by the facet class's name with an I prefixed. For example, if the facet by the name of AFacet were requested. Voyager would search for a class that implements the interface IAFacet.

Voyager searches objects for implementations of the facet's interface using the name of the primary object suffixed by the name of the facet. If a matching implementation is not found, Voyager searches using the primary object's superclasses. For example, if the primary object were of the class aClass (which extends java.lang.Object) and the facet AFacet were requested. Voyager would first look for a facet named aClassAFacet, then for ObjectAFacet, and finally AFacet.

When Voyager looks for an implementation for the facet, it searches the primary object's package and the facet's package.

### 3.3.2 Use of Dynamic Aggregation

Intelligent agents in the Multi-Agents Intrusion Detection System are lightweight in that they are oriented towards gathering and classifying data. The low-level agents themselves communicate directly only to their related data gathering agents and mediators. Adding low-level agent communication in the IDS allows agents to fuse related data in real time and take advantage of knowledge about the security status of related components in the system. Dynamic aggregation provides a convenient way of adding communication between low-level agents without adding excess baggage to the agents themselves.

Inter-agent communication was implemented by the use of sensitivity facets as an aggregate to the low-level agents. Sensitivity facets are a family of objects that communicate intrusion information among themselves and use information about related intrusions to affect future decisions about intrusions.

The issue of failed attempts to login provides a good example of how the sensitivity facets are used. A few failed logins in a distributed system are expected as users tend to forget passwords, try wrong passwords on the wrong systems or mistype as they try to login. However, in the face of an attack, a few failed logins may signal the next step in an intrusion as the attacker tries commonly-used or default passwords after having identified systems that allow virtual terminal or file transfer connections from the network. The failed logins sensitivity facets listen for network connection events from unusual sources (which occur as the attacker identifies her targets), remember these events, and increase their
sensitivity to failed logins for a period of time.

Agent collaboration in the IDS was not developed until after the agents had been designed, developed, and tested. Dynamic aggregation provided a way in which the agents could be extended without having to overload an agent with new features. However, the agents needed a bit of redesign to accommodate the sensitivity facets. Each adjustment facet must integrate itself into its primary agent to affect decisions on whether events are intrusive. The agents were modified to provide newly-obtained events to a listener (namely, the adjustment facet) for adjustment of the event's intrusion classification. The agents were also modified to notify a listener (the reporting facet) if an event was determined to be intrusive. This adjustment to the agent architecture makes the agents much more flexible and open to future enhancement by aggregation.

The reporting facets were designed to broadcast intrusion messages from their primary agent to other interested facets. The adjustment facets listen for intrusion messages from the reporting facets. Figure 3.3 illustrates the flow of information between the agents and facets.

![Diagram of Agent Architecture](image)

Figure 3.3 Facets for the low-level agents

### 3.3.3 Communication in the IDS

Auditing information is exchanged between agents using subclasses of an Audit class. The Audit class provides methods for serializing an audit object in a table of keys and values, deserializing an audit object, storing an audit object in a database, and obtaining an audit object from a database. Subclasses of the Audit class fully describe events like logins, changes to important files, network connections, and processes executed. Low-level agents obtain audit objects from data gathering agents and apply the first level of intrusion detection, using artificial intelligence or expert rules. Audit objects are passed vertically through the IDS (see Figure 3.1).
Intrusion information is exchanged between facets using descriptive strings in ASN.1 syntax. In the style advocated by Tim Bass [6], a small management information base (MIB) was designed to encapsulate information about intrusions. The MIB allows the low-level agents to communicate intrusion information at a higher, more abstract level and describe more knowledge about the intrusion. A sample of the ASN.1 hierarchy of intrusions follows.

**host** Intrusions identified on a host computer
- **host.priv_prog** Intrusions against a privileged program
- **host.auth** Intrusions related to authentication

**net** Intrusions related to network services
- **net.ip** Intrusions via the IP network protocol
- **net.ip.icmp** Intrusions via the ICMP protocol
- **net.ip.udp** Intrusions via the UDP protocol
- **net.ip.udp.service=nfs** Intrusions via NFS
- **net.ip.tcp** Intrusions via the TCP protocol
- **net.ip.tcp.service=login** Intrusions via the rlogin protocol

Listeners can register interest in any portion of the hierarchy by specifying a prefix. For example, a listener could register the prefix **net.ip.tcp** to listen for all TCP-related intrusions.

### 3.3.4 Communicating between Facets

The intrusions detected by each agent in the IDS were examined for related intrusions. Examples of such relations are shown in table 3.1.

Reporting facets were constructed to encode intrusion information in the MIB described previously and send the reports to interested facets. Adjusting facets encode the relationships between intrusions by registering interest in related intrusions. Adjustment facets interpret the received intrusion messages to affect their sensitivity level and use their sensitivity level to adjust intrusion classifications of events via their related low-level agent.

### 3.3.5 Implementation

In the following sections, the overall implementation of the prototype IDS is discussed. Then, the implementation of dynamic aggregation to add communications between agents is presented.
### Table 3.1 Agents and their related intrusions

<table>
<thead>
<tr>
<th>Agent</th>
<th>Related Intrusion</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Untrusted Connections:</td>
<td>Attack on network daemon: NetTCP</td>
<td>Attacker will often attempt to login via the network after attacking a network daemon or changing a configuration file</td>
</tr>
<tr>
<td></td>
<td>Changed configuration file</td>
<td></td>
</tr>
<tr>
<td>Critical Files</td>
<td>Attack on network daemon: NFS</td>
<td>Attacker may change a critical file to enable further attacks</td>
</tr>
<tr>
<td>Failed Logins</td>
<td>Port scanning</td>
<td>Attacker may scan available network ports to find machines on which to try to login</td>
</tr>
</tbody>
</table>

### 3.3.6 IDS Implementation

The prototype IDS has been implemented using Sun's Java Development Kit (JDK) version 1.1 [125] and ObjectSpace's Voyager Object Request Broker version 3 [101].

Java was chosen as the development language because of its platform independence, security, and speed of development. Java's platform independence has allowed us to compile and execute completely shared code on any one of several platforms including Silicon Graphic's IRIX and Hewlett-Packard's HP-UX commercial operating systems and free operating systems including FreeBSD and Linux. Java's security features include sandboxes for executing untrusted code, strict typing, bounds checking, bytecode verification, and code signing. Java's strict typing and object orientation assisted development by enforcing structure, allowing the addition of functionality by extending existing objects, and reducing time needed for debugging as compared to projects of similar complexity that have been implemented in the C language.

The runtime performance of Java code is a concern, but has not been an obstacle to the IDS project. The performance has been sufficient to demonstrate operation of the prototype IDS. When additional performance has been required, Just-In-Time (JIT) compilers that compile Java byte code into native machine code have satisfied the requirements.

The Voyager Object Request Broker was chosen for the prototype IDS because of its availability, development in Java as its native language, and support for mobile agents. Voyager provides mobility, message passing (independent of the target object's location), and naming services for Java objects, all of which are necessary for our prototype IDS. Voyager also supports extending lightweight agents via dynamic aggregation, or facets. Facets allow us to add capabilities to IDS agents, including agent communication and collaboration. We could also dynamically add countermeasures capabilities to agents. This would allow a running IDS system to adapt to new attacks by implementing new countermeasures.

The bottom tier of agents in the IDS is the set of stationary data cleaning agents. The stationary...
agents obtain information from sources including system logs, audit data, and operational statistics. The stationary agents convert the information into a common format for use by the higher levels of agents.

An example of a stationary agent is the DGFailedLoginAgent, which reads the system logs for reports of failed logins. It parses failed login messages from the system log into Login objects which describe when the failed login took place, which user account was used, and which computers were involved in the failed login.

The middle tier of agents in the IDS is the set of low level agents. Low level agents are mobile agents that gather information from the stationary agents. Low level agents process the gathered information to monitor and classify events. Low level agents then pass on the information to their mediator.

An example of a low level agent is the FailedLoginAgent, which visits each host’s DGFailedLogin Agent to retrieve the recent list of failed logins. If the number of failed logins in the entire distributed system exceeds a threshold in a short period of time, the failed logins are flagged as an attempted intrusion.

The prototype detects intrusions including attempts at trying multiple passwords on multiple machines, unusual TCP network connections, changes in critical files (with the assistance of Tripwire [58]), attacks against the sendmail mail transfer agent, and refused connections to unsafe network services.

Note, however, that the flow of information in this design is purely vertical (in terms of Figure 3.1) and agents at each level are not cooperating or coordinating with each other. The existing low-level agents are then considered “lightweight” in the view that they do not have the capability to communicate directly with each other.

### 3.3.7 Adding cooperation to the agents

Sensitivity facets were created to add communications capabilities to the low-level agents. The Sensitivity facets implement the adjustment and intrusion reporting functions shown in Figure 3.3. Mediators construct agents and attach Sensitività facets to agents. The Sensitivity facets implement the reporting and adjusting functions described in Section 3.3.3. As agents gather information, the information is passed through the Sensitivity facets for reporting and adjusting.

The interface ISensitivity was defined with the methods:

- **register()** To subscribe to events in the associated agent.
- **setIntrusionClasses()** To subscribe to interesting intrusions from other agents.
getSensitivity() To obtain the current sensitivity level.

sendIntrusionMessage() To publish messages about intrusions, and

recvIntrusionMessage() Through which intrusion messages are received by the facet.

A default implementation of the interface. Sensitivity was created. The register() method implementation subscribes to all events seen by its attached agent. The setIntrusionClasses() method implementation defines which intrusions are interesting to this facet. The getSensitivity() method implementation returns an indicator of how sensitive the facet is to suspicious events. The sendIntrusionMessage() method implementation publishes information about an intrusive event to other listening Sensitivity facets. The recvIntrusionMessage() method implementation receives messages about intrusive events from other Sensitivity facets and adjusts the facet’s sensitivity level based on the significance of the event.

Figure 3.4 shows a slice of the intrusion detection system with Sensitivity facets included. A specific Sensitivity facet, NetTCPAgentSensitivity, extends the basic Sensitivity facet to listen for intrusive activity including changes to files and buffer overflow attacks. If other agents report these intrusive activities, the sensitivity of the NetTCPAgent should be raised since the target may receive anomalous TCP connections in the near future which may be part of the intrusion. Because of Voyager’s rules for attaching facets to an agent, when a Sensitivity facet is added to the NetTCPAgent, the NetTCPAgentSensitivity facet is added to the NetTCPAgent.

The FailedLoginAgentSensitivity facet listens for intrusive activity reports from the NetTCPAgent. If the NetTCPAgent were to sense activity such as scanning for available telnet ports, the FailedLoginAgentSensitivity facet will raise its level of suspicion. An intruder may scan for available telnet ports, then connect to the discovered telnet ports and try to login to well-known accounts with typically-used passwords. If this were to happen, the FailedLoginAgentSensitivity facet would help detect the attack by lowering the threshold of acceptable failed logins.

### 3.3.8 Results

Intrusion events should be infrequent, so the intrusion detection system should rarely require the sensitivity capability. By only adding the sensitivity capability when it is required, the load on the network and speed of transmission of the agents is much better in the normal case.

In particular, the Java byte-compiled code for the FailedLoginAgent class is a total of 5665 bytes (2298 - 3367). The byte-compiled code for the FailedLoginAgentSensitivity class is a total of 5547
bytes (1540 - 4007). Permanently adding the sensitivity capability to the `FailedLoginAgent` would increase the size of the agent by 96%. Considering only the size of the code, the `FailedLoginAgent` is roughly half the size without the sensitivity facet as it would be with the sensitivity facet. At a constant transmission rate, the time to transmit the lightweight agent by itself is about half the time required to transmit the agent with the facet.

The savings are less dramatic for the `NetTCPAgent`. The Java byte-compiled code for the `NetTCPAgent` class is a total of 7434 bytes (4067 - 3367). The byte-compiled code for the `NetTCPAgentSensitivity` class is a total of 4493 bytes (486 - 4007). Permanently adding the sensitivity capability to the `NetTCPAgent` would increase its size by 60%.

![Diagram of the intrusion detection system with facets](image)

**Figure 3.4** A portion of the intrusion detection system with facets

### 3.3.9 Conclusions

Extending lightweight agents provides a convenient mechanism for implementing a new form of communication in our intrusion detection system. By developing facets that implemented listening and reporting functions, a significant new feature was added to the IDS without negatively affecting the existing agent design or operation. Operation of the system is improved in the normal case, since the load on the system due to the size of the agents is reduced when no intrusions are present.
3.3.10 Future Work

Extension of agents offers many possibilities for extending the intrusion detection system. A potential use for lightweight agent capabilities would be to add data mining capabilities to mediators in the intrusion detection system. A group of facets could be constructed that implement various data mining algorithms. Mediators could then dynamically add data mining algorithms for higher-level intrusion detection by adding appropriate facets.

The prototype MIB for intrusions should be formalized based on a taxonomy of intrusions. Using the ASN.1 syntax for describing intrusions seems to hold promise for integration of intrusion detection systems and communication between intrusion detection agents. Also, relationships between intrusions should be formally identified and encoded into facets. The prototype table of intrusion relationships was a first step in developing our sensitivity facets. Previous intrusion detection systems encoded these relationships as rules in an expert system [96]. Using agents and facets seems to offer a more dynamic, malleable way to deal with these relationships than encoding rules in an expert system.

An extension of the IDS system may be possible by using facets to implement data fusion to identify the source of an intrusion in real time. Some attacks identify the source IP address of the attacker, and at times even more information may be gathered from the source that help determine the identity of the attacker.

A possible further extension using facets would be to implement countermeasures to respond to intrusions. Facets could be designed that fuse intrusion information in real time. When the fused information meets certain criteria, the facets could direct the monitored system to take corrective or defensive action to deter or stop the intrusion. If the source of the intrusion is known (by the identification facets previously mentioned), countermeasures may be possible to prevent further attacks. Because facets are aggregated with their agents, the countermeasures facets would be operating on the system on which the actions would need to be executed. No additional communications overhead would be required to send commands to the target system to counter an attack.
4 LEARNING PREDICTIVE RULES FOR INTRUSION DETECTION

In this chapter, the design of a system call intrusion detector for the agent-based intrusion detection system is presented. We then explain our research in the application of artificial intelligence to the problem of identifying misuse of privileged programs. The Computer Immunology project [41] and the Java Agents for Meta-Learning project [67] explored the use of system call traces from privileged programs to detect intrusions. We use a feature vector approach to describe the system calls executed by a privileged program. We show that our feature vector representation works well for automated knowledge discovery using rule learning. We then employ feature subset selection using a genetic algorithm to reduce the number of features necessary in the feature vector and to improve the accuracy of the results by eliminating extraneous features.

4.1 Rule Learning from System Call Traces

Programs that provide network services in distributed computing systems often execute with special privileges. For example, the popular sendmail mail transfer agent operates with superuser privileges on UNIX systems. Privileged programs like sendmail are often a target for intrusions.

The trace of system calls executed by a program can identify whether an intrusion was mounted against a program [41][67]. Forrest’s project at the University of New Mexico [41] developed databases of system calls from normal and anomalous uses of privileged programs such as sendmail. Forrest’s system call data is a set of files consisting of lines giving a process ID number (PID) and system call number. The files are partitioned based on whether they show behavior of normal or anomalous use of the privileged sendmail program running on SunOS 4.1.

Forrest organized system call traces into sequence windows to provide context and showed that a database of known good sequence windows can be developed from a reasonably sized set of non-intrusive sendmail executions. Forrest then demonstrated that intrusive behavior can be determined by finding the percentage of system call sequences that do not match any of the known good sequences. The data sets that were used by Forrest’s project are available in electronic form on their Web site [39]. We use
the same data set to enable comparison with techniques used in related papers [67][132].

Our feature vector technique improves on Forrest's technique because it does not depend on a threshold percentage of abnormal sequences. Our feature vector technique compactly summarizes the vast data obtained from each process, enabling longer-term storage of the data for reference and analysis. With respect to other rule learning techniques, our approach induces a compact rule set that is easily carried in lightweight agents. Our technique also may mine knowledge from the data in a way that can be analyzed by experts.

Lee [67] used a portion of the data from Forrest's project to show that the RIPPER [26] learning algorithm could learn rules from system call sequence windows. Lee empirically found sequences of length 7 and 11 gave the best results in his experiments [67]. For training, each window is assigned a label of "normal" if it matches one of the good windows obtained from proper operations of sendmail; otherwise, the window is labeled as "abnormal". An example of the system call windows and labels are shown in Table 4.1.

<table>
<thead>
<tr>
<th>System Call Sequences</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>4. 2. 66. 66. 4. 138. 66</td>
<td>Normal</td>
</tr>
<tr>
<td>2. 66. 66. 4. 138. 66. 5</td>
<td>Normal</td>
</tr>
<tr>
<td>66. 66. 4. 138. 66. 5. 5</td>
<td>Normal</td>
</tr>
<tr>
<td>66. 4. 138. 66. 5. 5. 4</td>
<td>Abnormal</td>
</tr>
<tr>
<td>4. 138. 66. 5. 5. 4. 39</td>
<td>Abnormal</td>
</tr>
</tbody>
</table>

After RIPPER is trained, the learned rule set is applied to the testing data to generate classifications for each sequence window. Lee uses a window across the classifications of length $2L + 1$, where $L$ is the step size for the window, to group labels [67]. If the number of "abnormal" labels in the window exceeds $L$, the window is considered abnormal. An example of a single window over the classifications is shown in Table 4.2.

<table>
<thead>
<tr>
<th>RIPPER's Classification</th>
<th>System Call Sequences</th>
<th>Actual Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>4. 2. 66. 66. 4. 138. 66</td>
<td>Normal</td>
</tr>
<tr>
<td>Normal</td>
<td>2. 66. 66. 4. 138. 66. 5</td>
<td>Normal</td>
</tr>
<tr>
<td>Abnormal</td>
<td>66. 66. 4. 138. 66. 5. 5</td>
<td>Normal</td>
</tr>
<tr>
<td>Abnormal</td>
<td>66. 4. 138. 66. 5. 5. 4</td>
<td>Abnormal</td>
</tr>
<tr>
<td>Abnormal</td>
<td>4. 138. 66. 5. 5. 4. 39</td>
<td>Abnormal</td>
</tr>
</tbody>
</table>
The window scheme filters isolated noise due to occasional prediction errors. When an intrusion takes place, a cluster of system call sequences will usually be classified abnormal. In Table 4.2, since there are more "abnormal" classifications than "normal" in this window, then this entire window is labeled anomalous. Lee empirically found that values of $L = 3$ and $L = 5$ worked best for identifying intrusions [67].

Finally, when the window has passed over all the classifications, the percentage of abnormal regions is obtained by dividing the number of anomalous windows by the total number of windows. Lee uses this percentage to empirically derive a threshold that separates normal processes from anomalous processes. Warrender [132] uses a similar technique, the Locality Frame Count (LFC), that counts the number of mismatches in a group and considers the group anomalous if the count exceeds a threshold. Warrender's technique allows intrusion detection for long-running daemons, where an intrusion could be masked by a large number of normal windows with Lee's technique.

Lee [67] developed an alternate technique that predicts one of the system calls in a sequence. The alternate technique allows learning of normal behavior in the absence of anomalous data. Our technique requires anomalous data for training.

4.2 Representing System Call Traces with Feature Vectors

One of the goals of automated discovery of predictive rules for intrusion detection is to extract the relevant knowledge in a form that lends itself to further analysis by human experts. A natural question that was raised by examination of the rules learned by RIPPER [26] in Lee's experiments [67] and our experiments [48] was whether essentially the same performance could be achieved by an alternative approach that induced a smaller number of simpler rules.

To explore this question, we designed an alternative representation scheme for the data. This representation was inspired by the success of the "bag of words" representation of documents [118] that has been successfully used by several groups to train text classification systems [135]. In this representation, each document is represented using a vector whose elements correspond to words in the vocabulary. In the simplest case, the vectors are binary: a bit value of 1 indicates that the corresponding word appears in the document in question and bit value of 0 denotes the absence of the word.

In this experiment, the data was encoded as binary-valued bits in feature vectors. Each bit in the vector is used to indicate whether a known system call sequence appeared during the execution of a process. This encoding is similar in spirit to the "bag of words" representation for text documents.

Feature vectors were computed on a per-process basis from the sendmail system call traces [39].
Based on ideas from previous work [41][67], sequence windows of size 5 to 12 were evaluated for use with our feature vector approach. Sequence windows of size 7 were selected for their good performance in learning accuracy and relatively small dictionary size.

The training data was composed of 80% of the feature vectors randomly selected from normal traces and all of the feature vectors from the selected abnormal traces. To compare our results to those from the JAM project, four specific anomalous traces were selected for training. Five different selections of anomalous traces were also tested to ensure that arbitrarily selecting these four anomalous traces did not significantly affect the results.

The number of abnormal records in the training data was quite small (15 records) in proportion to the set of normal training data (520 records). To balance the weightings, the abnormal training data was duplicated 36 times so that 540 abnormal records were present in the training data. Lee [67] explains the rationale for balancing the data to obtain the desired results from RIPPER. From the feature vectors built from sequences of length 7, RIPPER efficiently learned a rule set containing seven simple rules:

\[
\begin{align*}
good & \text{ IF } a_{1406} = t \\
good & \text{ if } a_{67} = t \\
good & \text{ if } a_{65} = t \\
good & \text{ if } a_{576} = t \\
good & \text{ if } a_{132} = t \\
good & \text{ if } a_{1608} = t \\
bad & \text{ otherwise }
\end{align*}
\]

The size of this set of rules compares favorably to the set of 209 rules RIPPER learned when we used Lee’s system call window approach. The feature vector approach condenses information about an entire process’ history of execution. Feature vectors may make it easier for learning algorithms by aggregating information over the entire execution of a process rather than by looking at individual sequences.

Applying the learned rule set produced the results shown in Table 4.3. All traces except “Normal sendmail” are anomalous. Boldface traces were used for training. The total numbers of feature vectors, numbers of vectors predicted abnormal by RIPPER, and detection results are shown. Since a single feature vector represents each process, each trace tends to have few feature vectors.

The rules can not be expected to flag all of the processes in an attacked trace as an intrusion. While handling a mail message, sendmail spawns child processes that handle different parts of the
Table 4.3 Results of learning rules for feature vectors

<table>
<thead>
<tr>
<th>Trace Name</th>
<th>Total Feature Vectors</th>
<th>Vectors Predicted Abnormal</th>
<th>Attack Detected?</th>
</tr>
</thead>
<tbody>
<tr>
<td>chasin</td>
<td>6</td>
<td>3</td>
<td>Y</td>
</tr>
<tr>
<td>decode1</td>
<td>6</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>decode2</td>
<td>6</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>fwd-loops-1</td>
<td>2</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>fwd-loops-2</td>
<td>1</td>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>fwd-loops-3</td>
<td>2</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>fwd-loops-4</td>
<td>2</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>fwd-loops-5</td>
<td>3</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>recursive</td>
<td>25</td>
<td>23</td>
<td>Y</td>
</tr>
<tr>
<td>smtp65a</td>
<td>3</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>smtp5x</td>
<td>8</td>
<td>3</td>
<td>Y</td>
</tr>
<tr>
<td>smtp6hote</td>
<td>3</td>
<td>2</td>
<td>Y</td>
</tr>
<tr>
<td>sscp-1</td>
<td>1</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>sscp-2</td>
<td>1</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>sscp-3</td>
<td>1</td>
<td>1</td>
<td>Y</td>
</tr>
<tr>
<td>syslog-local-1</td>
<td>6</td>
<td>6</td>
<td>Y</td>
</tr>
<tr>
<td>syslog-local-2</td>
<td>6</td>
<td>6</td>
<td>Y</td>
</tr>
<tr>
<td>syslog-remote-1</td>
<td>7</td>
<td>7</td>
<td>Y</td>
</tr>
<tr>
<td>syslog-remote-2</td>
<td>4</td>
<td>4</td>
<td>Y</td>
</tr>
<tr>
<td>Normal sendmail (not used for training)</td>
<td>130</td>
<td>3</td>
<td>Y</td>
</tr>
</tbody>
</table>
procedures involved in receiving, queuing, and forwarding or delivering the message. Some of these processes involved in handling an intrusive transaction may be indistinguishable from processes handling a normal transaction because the attack only affects one of the processes. Therefore, if at least one of the processes involved in an intrusion is flagged as abnormal, we can identify the group of related processes as anomalous.

Several attacks did not result in successful intrusions. For our intrusion detection system, we identify all attacks, successful and unsuccessful, as intrusive activity that merits further investigation by the IDS. It would be unlikely that an attacker would attempt a single exploit and give up if it fails. The data mining portion of our intrusion detection system would then correlate these multiple (successful and unsuccessful) attacks.

The anomalous traces are clearly identified in our experiment with the exception of one of the minor intrusions, fwd-loops-2. The fwd-loop attacks are denial-of-service attacks where the sendmail process spends its time repeatedly forwarding the same message. The feature vector technique may need to be adjusted from simple binary values to statistical measures to identify this class of attack.

A benefit of the feature vector approach is the simplicity of the learned rules. Training takes place "off line" due to the amount of time need to learn a rule set. Each learned rule set for the sendmail system call feature vectors is simple: generally fewer than 10 rules, where each rule often consists of a conjunction of one or two Boolean terms. Such a small set of rules applied to this simple data structure should allow us to use this approach in a near real-time intrusion detection agent without placing an excessive load on a system. A small, simple rule set also may lend itself to human expert examination and analysis in data mining situations [10].

Another benefit of the feature vector approach is the condensed representation of a process by its fixed-length feature vector. The list of system calls executed by a process can be enormous. Storing this information in its entirety is infeasible. Representing the data by a relatively short fixed-length string helps solve the problems of transmitting and storing the data. This technique realizes the mobile agent architecture's goal of reducing and summarizing data at the point of generation.

4.3 Feature Subset Selection Using Genetic Algorithms

A learning algorithm's performance in terms of learning time, classification accuracy on test data, and comprehensibility of the learned rules often depends on the features or attributes used to represent the examples. Feature subset selection has been shown to improve the performance of a learning algorithm and reduce the effort and amount of data required for machine learning on a broad range
of problems [77]. A discussion of alternative approaches to feature subset selection can be found in [57][134][77].

The benefits and affects of feature subset selection include:

- Feature subset selection affects the accuracy of a learning algorithm because the features of a data set represent a language. If the language is not expressive enough, the accuracy of any learning algorithm is adversely affected.

- Feature subset selection reduces the computational effort required by a learning algorithm. The size of the search space depends on the features: reducing the feature set to exclude irrelevant features reduces the size of the search space and thus reduces the learning effort.

- The number of examples required to learn a classification function depends on the number of features [66][94]. More features require more examples to learn a classification function to a desired accuracy.

- Feature subset selection can also result in lower cost of classification (because of the cost of obtaining feature values through measurement or simply the computation overhead of processing the features).

Against this background, it is natural to consider feature subset selection as a possible means of improving the performance of machine learning algorithms for intrusion detection [42].

Genetic algorithms and related approaches [44][85][60] offer an attractive alternative to exhaustive search (which is infeasible in most cases due to its computational complexity). They also have an advantage over commonly used heuristic search algorithms that rely on the monotonicity assumption (i.e., addition of features does not worsen classification accuracy) which is often violated in practice [134].

The genetic algorithm for feature subset selection starts with a randomly generated population of individuals, where each individual corresponds to a candidate feature subset. Each individual is encoded as a string of 0's and 1's. The number of bits in the string is equal to the total number of features. A 1 in the bit string indicates an attribute is to be used for training, and a 0 indicates that the attribute should not be used for training. The fitness of a feature subset is measured by the test accuracy (or cross-validation accuracy of the classifier learned using the feature subset) and any other criteria of interest (e.g., number of features used, the complexity of the rules learned).

We used the RIPPER rule learning algorithm as the classifier. The training data is provided to RIPPER, which learns a rule set from the data. The number of conditions in the learned rule set is
counted, and this value is used to determine the complexity of the learned hypothesis. The learned rule set is applied to the test examples and the determined accuracy is returned to the feature subset selection routine. The fitness of the individual is calculated, based on the accuracy of the learned hypothesis \( \text{accuracy}(x) \), the number of attributes \( \text{cost}(x) \) used in learning, the complexity of the learned hypothesis \( \text{complexity}(x) \), and weights \( w_{\text{accuracy}}, w_{\text{cost}}, w_{\text{complexity}} \) for each parameter:

\[
\text{fitness}(x) = w_{\text{accuracy}} \times \text{accuracy}(x) + w_{\text{cost}} \times \text{cost}(x) + w_{\text{complexity}} \times \text{complexity}(x)
\]

This fitness is then used to rank the individuals for selection. Other methods of computing fitness are possible and are discussed by Yang [134].

A primary goal in using feature subset selection on this intrusion detection problem is to improve accuracy. A high percentage of the intrusion detection alerts reported by current intrusion detection systems are false alarms. Our system needs to be highly reliable, and we would like to keep false alarms to a minimum. A secondary goal is to reduce the amount of data that must be obtained from running processes and classified. This would reduce the overhead of our intrusion detection approach on the monitored system.

### 4.3.1 Feature Subset Selection Results

The genetic algorithm used standard mutation and crossover operators with 0.001 probability of mutation and 0.6 probability of crossover with rank-based selection [44]. The probability of selecting the best individual was 0.6. A population size of 50 was used and each run went through 5 generations.

We started with the training data used for the previous feature vector experiment (1060 feature vectors). We added an additional copy of each unique feature vector in the training data (72 feature vectors) to ensure that rare but potentially important cases had a reasonable probability of being sampled in the training and testing phases. This gave a total of 1132 feature vectors in the input to the genetic algorithm.

To show the general effectiveness of genetic feature selection on this problem, Table 4.4 shows the results of five separate runs of the genetic algorithm with RIPPER with identical parameters used for each run. The number of attributes is significantly reduced while the accuracy is maintained.

Table 4.5 shows the results of using the rules from the best individuals found in the five genetic feature selection runs and compares the results to the original results learned from all the features. All traces except “Normal sendmail” are intrusions. Boldface traces were used for training. Despite using only about half the features in the original data set, the performance of the learned rules was
Table 4.4 Feature subset selection results with constant genetic parameters

<table>
<thead>
<tr>
<th>Trial</th>
<th>Training Accuracy of Best Individual</th>
<th>Attributes Used by Best Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>98.9399</td>
<td>847</td>
</tr>
<tr>
<td>2</td>
<td>98.8516</td>
<td>857</td>
</tr>
<tr>
<td>3</td>
<td>99.1166</td>
<td>846</td>
</tr>
<tr>
<td>4</td>
<td>99.1166</td>
<td>849</td>
</tr>
<tr>
<td>5</td>
<td>99.1166</td>
<td>839</td>
</tr>
</tbody>
</table>

comparable to that obtained using the entire set of features. After feature subset selection, none of the feature vectors from normal sendmail are labeled as abnormal. This shows an improvement in the rate of false positives.

Table 4.5 Results from rules learned by genetic feature selection

<table>
<thead>
<tr>
<th>Trace</th>
<th>All Attributes</th>
<th>Trial 1</th>
<th>Trial 2</th>
<th>Trial 3</th>
<th>Trial 4</th>
<th>Trial 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>chasin</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>decode1</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>decode2</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>fwd-loops-1</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>fwd-loops-2</td>
<td>X</td>
<td>X</td>
<td>Y</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>fwd-loops-3</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>fwd-loops-4</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>fwd-loops-5</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>recursive</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>sm565a</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>sm5x</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>smndhole</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>sscp-1</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>sscp-2</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>sscp-3</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>syslog-local-1</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>syslog-local-2</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>syslog-remote-1</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>syslog-remote-2</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Normal sendmail</td>
<td>120</td>
<td>0</td>
<td>120</td>
<td>0</td>
<td>120</td>
<td>0</td>
</tr>
</tbody>
</table>

4.4 Analysis

A comparison of the effectiveness of RIPPER on the problem using two different data representations and genetic feature selection algorithm follows.

Table 4.6 illustrates the advantages of the feature vector representation over the system call windows for this learning problem. The feature vector representation allows the learning algorithm to learn a hypothesis much faster and with comparable accuracy on the normal test data, and the complexity of
the hypothesis is much smaller. Using genetic feature selection on the feature vectors is time consuming but further improves the learned hypothesis and reduces the set of attributes used for learning.

### 4.4.1 Rules Learned by RIPPER

An example set of rules that were learned in first trial of RIPPER with genetic feature subset selection is shown below:

- good IF a1024 = t .
- good IF a27 = t .
- good IF a873 = f AND a130 = f .
- good IF a12 = t .
- good IF a191 = t .
- good IF a223 = t .
- good IF a327 = t .
- bad IF .

The set above contains 8 individual rules composed of 8 tests. which correspond to this pseudocode:

IF "unlink,close,unlink,unlink,close,_gettimeofday,open" seen THEN good
ELSE IF "chmod,ioctl,fstat,write,close,unlink,rename" seen THEN good
ELSE IF "sigsetmask,sigblock,sigvec,sigvec,sigsetmask,sigblock,sigvec" not seen and
   "close,setitimer,close, gettimeofday,link,socket,fcntl" not seen THEN good
ELSE IF "accept,wait4,wait4,wait4,wait4,accept,fork" seen THEN good
ELSE IF "fcntl, gettimeofday, getpid, sendto, accept, fork, close" seen THEN good
ELSE IF "fstat, mmap, close, open, fstat, mmap, getdents" seen THEN good
ELSE IF "getpid, sendto, accept, wait4, wait4, accept, close" seen THEN good
ELSE bad

Each of the rule sets from the five genetic algorithm trials contains rules that can be found in the other
rule sets. The third and fourth trials contain mostly unique rules, while the other three runs contain
a majority of rules that are duplicated in other rule sets. The similarities of rules between runs likely
indicates the strength of particular sequences in identifying normal behavior.

Because the rule sets identify normal processes and consider all others abnormal, none of the rules
identifies particular abnormal system call sequences. Consequently, the rules do not identify system call
sequences that would directly signal an intrusion. However, these rules may lead to an understanding
of how an attack causes the typical sequence of system calls to change.

In general, the small size of the rules sets learned by RIPPER from the system call feature vectors
and the performance of these learned rule sets indicates that a concise set of rules clearly distinguish
normal sendmail processes from anomalous.

4.5 Conclusion and Future Work

Intrusion detection and abuse detection in computer systems in networked environments is a problem
of great practical interest. This chapter investigated the classification of system call traces for intrusion
detection through the technique of describing each process by a feature vector. From the feature vector
representation RIPPER learned a small, concise set of rules that was successful at classifying intrusions.
In comparison with other techniques, the feature vector representation does not depend on thresholds to
separate normal from anomalous. We are concerned that establishing an arbitrary threshold is difficult
and would require tuning in practice to balance false alarms (false positives) against missed intrusions
(false negatives).

The rule sets learned using the feature vector representation are an order of magnitude simpler than
those obtained using other approaches reported in the literature [48][67]. This is especially noteworthy
given the fact that all of the experiments in question used the same rule learning algorithm. We conjecture that the feature vector representation used in our experiments is primarily responsible for the differences in the rule sets that are learned. The feature vectors condense information from the entire execution of a process compared to the fine-grained detail of individual sequences. The scope of information contained in the feature vectors may make it easier for learning algorithms to learn simple rules.

It was further shown that feature subset selection reduced the number of features in the data, which resulted in less data and effort required for training due to the smaller search space. Feature selection also gave equivalent accuracy with a smaller set of features.

We have integrated the learned rules into a mobile agent running on a distributed system consisting of Pentium II systems running FreeBSD. This laboratory network is connected by a firewall to the Department of Computer Science’s network so we may operate the intrusion detection system in a controlled environment. For operation of the IDS, a Voyager server is started on each host in the monitored distributed system. The mobile agent is travels through the system, classifies sample sendmail system call feature vectors, and reports the results to its mediator. The mediator reports the results to the user interface and optionally stores the information in a database for potential mining and warehousing operations. We have implemented a set of Java classes that can interpret and apply the RIPPER rules, which allows our mobile agent to bring its classifier and rule set(s) with it as it travels through the distributed system.

Open issues include the use of this technique in heterogeneous distributed systems. Specific rule sets may need to be developed for each node in a distributed system due to variabilities between operating systems and workload characteristics. Fortunately, the rule sets discovered by RIPPER have been small, so mobile agents ought to be able to carry multiple rule sets without becoming overly “heavy”.

Another issue is whether this technique could be applied in real time. Feature subset selection itself is computationally expensive, so training and refining the agent can not be done in real time. After the agent is trained, our technique can determine whether a process is an intruder only after the process has finished, which provides near real time detection. Either Warrender’s [132] or Lee’s techniques [67] would allow anomaly detection in real time during the execution of the process. Our technique could be refined to determine the likelihood that a process is intrusive during the process’ execution, giving real time detection. This refinement would be necessary for long-lived daemons such as HTTP servers.

We would also like to know how well this technique applies to privileged programs other than sendmail. Warrender worked with five distinct privileged programs and identified cases where different
thresholds and or different algorithms worked better for different programs [132]. Based on her work, we expect this technique will be successful for other programs than sendmail.

Work in progress on intrusion detection is aimed at the integration of data-driven knowledge discovery agents into a distributed knowledge network for monitoring and protection of distributed computing systems and information infrastructures. The investigation of machine learning approaches to discover patterns of coordinated intrusions on a system wherein individual intrusions are spread over space and time is of particular interest in this context.
5 A SOFTWARE FAULT TREE APPROACH TO MODELING INTRUSIONS

This chapter examines the use of a software safety technique, Software Fault Tree Analysis (SFTA), for use as a model for intrusions. Benefits of using SFTA for intrusions are considered. Intrusions are categorized by temporal stages, and Software Fault Tree (SFT) models of intrusions in each stage are developed. Paths through the fault trees are examined for two sample intrusions.

Benefits of the SFTA technique include developing an understanding of intrusions, determining which events in an intrusion may be detected, assisting the development of an intrusion detection system, and testing the intrusion detection system.

5.1 Introduction

Software Fault Tree Analysis (SFTA) is a method for identifying and documenting the combinations of lower-level software events that allow a top-level event (or root node) to occur. When the root node is a hazard, the SFTA assists in the requirements process by describing the ways in which the system can reach that unsafe state [68]. The safety requirements for the system can then be derived from the software fault tree, either indirectly [29][78] or directly via a shared model such as a dynamic systems model used by Hansen to link SFTA with a requirements specification [45].

In the work described here, we use SFTA to assist in determining and verifying the requirements for an intrusion detection system. The root node of the top-level SFTA is not strictly a hazard, as in a safety analysis, but an intrusion which is a violation of a system's security policy. Intrusions result in compromise of exclusivity (unauthorized disclosure of data or use of services), integrity (unauthorized modification of data), or availability (denial of service). Whereas safety failures are often accidental or unexpected, intrusions are intentional, perpetrated by individuals, and can be expected to occur. Both safety and security failures represent potentially significant or catastrophic losses.

Intrusions can occur in a variety of ways. The software fault tree models the combinations and sequences of events by which intrusions can occur. The understanding and capture of domain knowl-
edge needed to accurately define the requirements on an Intrusion Detection System (IDS) is difficult. Questions such as what intrusions can be feasibly detected by the IDS software, at what stage of an intrusion the IDS software should detect each attack, and what assurances can be given that the IDS software detects intrusions must be addressed by the requirements analysis. The goal is not to build a system in which the root node never occurs, but to build an IDS in which the root node never occurs undetected.

We are investigating formal underpinnings for IDSs in terms how to describe intrusions, identify intrusion characteristics, and provably detect intrusions based on observable characteristics. Existing intrusion detection systems tend to be built by selecting a set of data sources and developing a classification system to identify some set of intrusions using the selected data [48]. A broader approach that begins with a thorough analysis of intrusions and supports development of a theoretical model of intrusion detection will answer questions about which intrusions are detectable, how they can be detected, how the data from different sensors may be correlated, and a quantifiable certainty that an intrusion took place.

The field of software safety provides techniques for developing answers to the problems posed above. Software safety is concerned with software-related errors that represent “potentially significant or catastrophic failure effects” [79]. Leveson has described software safety engineering techniques in her book Safeware [68], and Leveson and others have reported on the application of safety engineering techniques to a number of software systems [70][71][79][80]. When safety engineering techniques are applied to safety-critical software systems, potentially significant or catastrophic errors are consistently identified. Ideally, safety engineering techniques are applied as early in the software development process as possible, such as in the requirements specification phase of a project [80].

The nature of software safety-related failures is different from intrusions. Safety failures result in loss of equipment or property as well as cause or contribute to death, injury, or environmental harm [69]. Intrusions result in compromise of exclusivity (unauthorized disclosure of data or use of services), integrity (unauthorized modification of data), or availability (denial of service). Safety failures are usually accidental and unexpected, while intrusions are intentional, perpetrated by individuals, and may be anticipated. However, the analogies between safety and security are strong enough to merit the application of software safety techniques to intrusion detection. Both safety and security failures represent potentially significant or catastrophic losses: for a safety-critical system that requires strong security, the two domains become intertwined. Rushby further relates safety and security as both being components of “critical systems” and promotes the use of hazard analysis [116].
Engineering-based analysis of the intrusion domain assists the requirements specification and design of an intrusion detection system. Safety engineering techniques may be applied to the analysis of the intrusion and intrusion detection domains. An in-depth analysis of intrusions using known defects in software and protocol specifications may lead to a verifiable specification of detectors for the analyzed intrusions.

5.2 Software Fault Tree Analysis

The Software Fault Tree Analysis used to model the intrusions is a backward search. It begins with a known hazard (here, an intrusion) as the root node and traces back through the possible parallel and serial combinations of events that caused such an intrusion. The fault tree graphically represents this information in a diagram of events and logic gates leading to each hazard. Normally, the goals of developing a software fault tree include identifying contributing circumstances, demonstrating that a system may not reach an unsafe state or unsafe states are reached with very low probability, or determining ways to recover from paths to unsafe states [69]. In the intrusion domain, however, widely-deployed existing systems and protocols which are unsafe (i.e., allow intrusions) are modeled in the intrusion detection domain to enable reasoning about the possible & necessary combinations of events that lead to intrusions. Such analysis leads to advances examined in section 5.4.3.

Figure 5.1 shows commonly-used fault tree symbols. The procedure for fault tree analysis starts with identifying a hazard. The hazard becomes the root of the fault tree. Necessary preconditions for the hazard are specified in the next level of the tree and joined to the root with a logical And or a logical Or. Each precondition is similarly expanded until all leaves are events that occur with some calculable probability or cannot be further analyzed. Fault tree analysis is used at the system level to identify high-level requirements for software safety. Software fault tree analysis is then performed on code, designs, or requirements specifications [79].

5.2.1 Minimum cuts

Definitions.

2.1 A fault tree is a rooted acyclic digraph with vertices including AND gates, OR gates, and events.

2.2 A cut set is a set of basic events whose occurrence causes the system to fail [111]. More formally, a cut $C$ of a fault tree $G = (V, E)$ with root $r$ is a subset $C$ of $V$ such that:

1. $r \in C$. 

Rectangle indicates an event to be analyzed further.

Circle represents a basic fault event or primary failure of a component. It requires no further development.

House is used for events which normally occur in the system. It represents the continued operation of the component.

Diamond is used for non-primal events which are not developed further for lack of information or insufficient consequences.

Oval indicates a condition. It defines the state of the system that permits a fault sequence to occur. It may be normal or result from failures.

AND gate indicates that all input events are required to cause the output event.

OR gate indicates that one or more of the input events are required to produce the output event.

Figure 5.1 Relevant fault tree symbols
2. \( \forall \text{ AND gate node } a \in C. \forall (a, a_e) \in E. a_e \in C. \) and

3. \( \forall \text{ OR gate node } o \in C. \exists (o, o_e) \in E. o_e \in C. \)

2.3 A cut set is called a **minimum cut set** if it cannot be reduced and can still cause the system to fail [111]. Equivalently, a minimum cut \( M \) of a fault tree \( G = (V, E) \) is a cut of \( G \) such that \( \forall M' \subseteq M. M' \) is not a cut of \( G \).

2.4 A **use case** is a high-level description of a user's interaction with a system.

A **minimum cut** of a fault tree gives a minimum set of successful events necessary to satisfy the root. A minimum cut of an intrusion fault tree describes a scenario in which an attacker successfully exploits security flaws to achieve the goal of compromising the system. Manian et al. [81] use Binary Decision Diagrams as an alternative to cutset-based solutions of fault trees for large, combinatorial solutions. In our current work, the size of the fault trees has been manageable using traditional cutset-based solutions.

5.3 Developing Fault Trees for Intrusions

Intrusion fault tree modeling draws from a variety of sources. The standards used in current TCP, IP networks are publicly available. Proposals and standards for IP networks are published by the Internet Engineering Task Force (IETF) as Requests for Comment (RFC's) and Standards (STD's). Implementations of most network protocols are freely available in software such as Linux, FreeBSD, and Apache, allowing public review for security issues. Numerous researchers and hackers actively discover and publish security vulnerabilities in public forums including mail lists such as bugtraq and web sites such as [www.securityfocus.com](http://www.securityfocus.com).

Faults that are generally UNIX-centric are considered in the fault trees, although many similar problems (e.g. buffer overflows) exist in software on other systems. Rather than looking directly at the source code for these systems, the immense body of publicly-discussed vulnerability information is used as input for development of the sample fault trees discussed here.

5.3.1 Reasonable Fault Trees

Each complete, successful intrusion can vary greatly from other intrusions, and analysis of complete intrusions is difficult. A monolithic fault tree that would attempt to describe all attacks would be huge, unwieldy, and less useful than several trees divided in a systematic manner. A reasonable approach is
to divide intrusions into stages of attacks that achieve intermediate goals of the attacker and develop
fault trees that model each of the stages.

Ruiu's analysis of intrusions [115] divides intrusions into seven stages:

1. Reconnaissance - find targets
2. Vulnerability Identification - find vulnerabilities in the targets
3. Penetration - gain unauthorized access to a target through a vulnerability
4. Control - gain privileges over the target
5. Embedding - hide activity within the target and ensure future access to target
6. Data Extraction & Modification - obtain or change confidential information
7. Attack Relay - attack other targets

Our analysis of documented intrusions, such as the intrusion into www.apache.org [129], and our
hypothesized intrusions correspond to Ruiu's breakdown.

5.3.2 Developed Examples of SFT for Intrusions

The OR gates in the fault trees shown are "true" if any input is true. The AND gates are "true" if
all inputs are true in the current context, where the context may be a virtual network connection, a
user's login session, a series of related transactions, or some other temporal context. The AND gates
in the fault trees shown generally assume that the child events occur in left-to-right order. (Hansen et
al. [45] discuss the ambiguities of traditionally accepted fault trees.)

The developed sample fault trees are not complete, but represent a significant subset of the intrusions
of most concern to administrators of distributed networks. Examples of how paths in the trees can
describe complete intrusions is discussed below.

We consider an active attack to be an attack that includes events that can be seen within the
distributed system under the organization's control. The majority of attacks in an intrusion are active.

We consider a passive attack to be an attack that leaves no trace within an organization's distributed
system. Examples of passive attacks are password sniffing or DNS zone transfers from an off-site
secondary DNS server.
Figure 5.2 Reconnaissance fault tree
5.3.2.1 Reconnaissance

The reconnaissance phase identifies potential targets within an organization's networks. Network targets include not only multiuser hosts (e.g., UNIX or Windows NT systems) but also routers, intelligent hubs, and perhaps even modems. The services offered by systems and names of users on the systems are also useful bits of information for an attacker, but only a portion of this information may be required to mount a successful attack. Figure 5.2 shows a sample fault tree for the reconnaissance phase.

5.3.2.2 Vulnerability Identification

Vulnerability identification is closely related to reconnaissance. In this phase, an attacker searches for vulnerabilities that can lead to penetration. The attacker sequentially scans many ports looking for services known to be vulnerable to attack. Port scanning is a "noisy" active attack, and is usually easy to detect unless done very slowly.

Figure 5.3 matches searches for vulnerable versions of servers for numerous services and for the existence of "remote control" services such as Back Orifice and NetBus. "Remote control" services allow the remote operation of a computer and observation of the activities of the legitimate user.

5.3.2.3 Penetration

Penetration occurs when an attacker obtains unauthorized access to a system. Penetration methods include exploitation of various network server daemon vulnerabilities (poor authentication and buffer overflows), authenticating with illicitly obtained passwords, and TCP session hijacking. Figures 5.4, 5.5, 5.6, and 5.7 together represent a sample fault tree for the penetration stage of intrusions.

5.3.2.4 Control

An attacker needs to gain sufficient privilege in a system to continue to the next stages of the intrusion. Often an attacker must obtain privileges equivalent to those of the system administrator to gain sufficient control of a system. If the penetration was particularly effective and sufficient privilege was already gained, this step may not be necessary.

Mechanisms traced in figure 5.8 include exploiting buffer overflows in privileged local programs, exploiting race conditions in temporary files or signals, exploiting weak permissions on critical files and devices, and cracking a password for an administrator's account.
Figure 5.3 Vulnerability identification fault tree
Figure 5.5 Penetration fault tree: Using buffer overflows in network daemons
Figure 5.6 Penetration fault tree: Gaining access by modifying configuration files
Figure 5.7 Penetration fault tree: Gaining access through abuse of authentication methods
Figure 5.8 Control fault tree
5.3.2.5 Embedding

Embedding involves the installation or modification of a system so that even if the attacker is discovered and steps are taken to recover the system, the attacker will still be able to enter the system. For example, the system bootstrap code could be modified to re-insert backdoors if the system executable programs are restored from backups or installation media. Typical techniques include installing Trojan horses, backdoors, and other rootkit programs, removing traces of the intrusion from system logs, and disabling detection systems.

Figure 5.9 identifies two different rootkit installations by matching particular sets of modified system files. A rootkit is a collection of embedding programs that allow an attacker to hide his activities and may include programs for use in the next step, data extraction & modification. The two rootkits identified are the Linux rootkit from 1994 and version 4 of the Linux rootkit.

The 1994 Linux rootkit is very basic. It is comprised of four programs:

fix A program to install the other rootkit programs in place of the legitimate system programs while copying the permissions, time, date, and checksum of the legitimate program to the rootkit program.

login A modified version of the login user authentication program that allows anyone to login with the password wh00t!.

netstat A modified version of the netstat network activity reporting program that hides network activity by particular users or connections to particular hosts or ports.

ps A modified version of the ps process status reporting program that hides processes owned by particular users, processes executing on particular terminals, and processes executing particular programs.

Version 4 of the Linux rootkit installs a much larger set of embedding programs than the 1994 rootkit, including chfn, chsh, crontab, find, inetd, passwd, rshd, syslogd, tcp wrappers, and a password sniffer.

5.3.2.6 Data Extraction & Modification

In the data extraction and modification phase, the attacker gathers information about the configuration and operation of the system. Covert channels may be used to move discovered data from the
Figure 5.9 Embedding fault tree
compromised system to the attacker's base. An example of useful extracted data would be cracked account passwords.

### 5.3.2.7 Attack Relay

After a system is fully compromised, it may be used for attack relaying. Attacks can be launched against affiliated (trusting) hosts to expand the number of hosts under the attacker's control. A system also may simply be used to participate in distributed denial of service attacks [35][33][34][32]. Figure 5.10 represents some basic faults seen from Stacheldraht. Tribe Flood Network, and Trinoo distributed denial of service attacks. For example, the Stacheldraht distributed denial of service attack depends on relay agents that are installed by an attacker on compromised computers. The Stacheldraht relay agent accepts commands from a handler, which manages a group of relay agents. The handler, in turn, is controlled by the Stacheldraht client which provides the user interface for the attacker. The Stacheldraht relay agent implements ICMP flood, SYN flood, UDP flood, and "Smurf" attacks [18][23]. With a number of Stacheldraht agents distributed logically in the Internet, an attacker can mount an attack that can completely disable the Internet access of even a large, well-connected organization.

Many other forms of attack relaying exist, including automated and manual means. Stacheldraht, Tribe Flood Network, and Trinoo are a sample of the current well-known, well-analyzed distributed attack relaying systems.

### 5.4 Experience with Fault Trees for Intrusions

The relationship of the developed fault trees to two actual intrusions is examined in this section. Each intrusion follows one of the multiple paths through each of the staged subtrees in Figures 5.2–5.9. A portion of the fault tree of Figure 5.7, describing the FTP bounce attack, was selected for further analysis. The FTP bounce attack subtree is particularly interesting because it shows several time-ordered steps which must take place for the attack to be successful.

#### 5.4.1 Example 1: FTP SITE EXEC Intrusion

The SITE EXEC attack against the wuftp daemon [25] is an interesting buffer overflow exploit. When someone logs into the wuftp daemon as the user anonymous or ftp, the daemon requests the email address be entered as the password. However, an attacker can send malicious shell code in response to the password prompt. Then, if the SITE EXEC command is enabled, the attacker can send a SITE EXEC command with %-formatting characters that cause a buffer to overflow with data previously
ATTACK-RELAY - Attack Relay phase of an intrusion

Figure 5.10 Attack relay fault tree
obtained as the password [99]. An example of an attacker’s FTP SITE EXEC command might be the string “SITE EXEC ..(object code to execute)%n(more object code)%n(more object code)%n\n”. The %n formatting sequence is interpreted by the printf function as a request to write the number of characters output so far into an integer location. A patient attacker can develop an exploit that uses the %n formatting sequence to overwrite a procedure’s return address on the stack. When the printf function returns, and if the exploit is successful, the code provided by the attacker is executed. In the case of the wuftpdp FTP server, the string provided as the argument to the SITE EXEC command is passed to the syslog function, which interprets %-formatting characters using the printf family of routines.

5.4.1.1 Reconnaissance

The intruder discovers the FTP server host by using any one or more of the methods under the “HostDiscovery” node in the reconnaissance tree in Figure 5.2.

The intruder also discovers the availability of the FTP server by one of the methods under the “TCP-Service-Discovery” node in the reconnaissance tree.

5.4.1.2 Vulnerability Identification

The intruder may or may not take the time to make a connection to the FTP server and verify that the version number reported by the server is vulnerable to the FTP SITE EXEC attack. (Known FTP vulnerabilities have not yet been expanded under the “FTPD” node in the vulnerability identification tree in Figure 5.3.)

5.4.1.3 Penetration

The intruder connects to the FTP server, gives “anonymous” or “ftp” as the user name, and enters malicious shell code as the password. The intruder then issues a SITE EXEC command containing printf-style formatting character sequences which overflow the character buffer on the process’ stack to executed the code provided as the previously-entered “password.” If the overflow is successful, the code provided by the intruder is executed with root privileges. These activities match the penetration subtree of figure 5.5.

The successful FTP SITE EXEC attack also gives the attacker control, so the attacker can move on to the later stages of the intrusion.
5.4.1.4 Control

We assume the successful penetration results in privileged access, so the control phase of the intrusion may be by-passed.

5.4.1.5 Embedding

The intruder installs the Linux Rootkit version 4, which replaces a number of programs with Trojaned implementations that hide the attacker's activities. Figure 5.9 matches the changes to the file system that result from the installation of the Linux Rootkit version 4.

5.4.1.6 Data Extraction

The intruder installs and runs a password sniffer that takes user names and passwords from telnet and ftp sessions on the LAN.

5.4.1.7 Attack Relay

The intruder installs and runs a distributed denial of service agent, such as Trinoo, TFN, or Stacheldraht. The intruder can then use the system to execute attacks against other networked sites.

5.4.1.8 Derived IDS Requirements

The software fault trees involved in this intrusion helped identify the software requirements for the mobile agent software tasked with detecting the FTP SITE EXEC intrusion. Examination of the trees shows that in the penetration subtree's path taken by the FTP SITE EXEC attack, it is feasible to detect this penetration with software. An intrusion detection system should monitor PASS commands in an FTP session for data that does not represent a valid sequence of printable characters. That is, an invalid sequence of printable characters is a minimum cut event. The analysis does not say anything about how the monitoring should be implemented or performed; it merely leads to requirements for the intrusion detection system.

5.4.2 Example 2: FTP Bounce Intrusion

The FTP bounce attack can be used to transfer data to a network port to which an attacker does not normally have access [20]. One way to exploit this vulnerability is to send data to a remote shell server that trusts the FTP host via the FTP server. After an attacker discovers an FTP server and a host running rsh that might trust the FTP server, the attacker tries this exploit:
1. Uploads a specially-formatted file to the FTP server:

2. Issues an FTP PORT command that directs the FTP server to send its next download to port 514 on the target host:

3. Issues an FTP GET command to “download” the contents of the previously-uploaded file into port 514 on the target: the GET command opens a connection from the FTP server on port 20 to the rsh daemon on the target.

4. If the target trusts the FTP server, the rsh daemon will accept the contents of the file as if it were user input and execute the given command.

The following steps in an intrusion based on an FTP bounce attack show how the trees fit the entire intrusion.

5.4.2.1 Reconnaissance

The intruder discovers the FTP server host and target host, using any one or more of the methods under the “HostDiscovery” node in the reconnaissance tree.

The intruder also discovers the availability of the FTP server and RSH server by one of the methods under the “TCP-Service-Discovery” node in the reconnaissance tree.

5.4.2.2 Vulnerability Identification

The intruder may or may not take the time to make a connection to the FTP server and verify that the version number reported by the server is vulnerable to the FTP bounce attack. The intruder also needs a directory on the FTP server to which he may upload a file: if the intruder has no access to the FTP server other than “anonymous”, the intruder will have to search for such a directory. (The large number of known FTP vulnerabilities has not yet been expanded under the “FTPd” node in the vulnerability identification tree.)

The intruder will likely have to assume that the target host trusts the FTP server host, unless the intruder already has some access to the target host and can read the etc:hosts.equiv or “root..rhosts” files. We have not considered “insider access” in the vulnerability identification tree.

5.4.2.3 Penetration

The intruder uploads the shell command file to the FTP server and issues the appropriate FTP commands to cause the FTP server to “download” the file into the target’s RSH service. The “FTP-
Bounce" subtree of figure 5.7 shows the required FTP commands and responses. The structure of the subtree enforces the order of the events in the FTP command response stream.

The successful FTP bounce attack mounted against a privileged account on the target also gives the attacker control, so the attacker can move on to the later stages of the intrusion.

5.4.2.4 Control

We assume the successful penetration results in privileged access, so the control phase of the intrusion may be bypassed.

5.4.2.5 Embedding

The intruder installs the Linux rootkit version 4, which replaces a number of programs with Trojaned implementations that hide the attacker’s activities. Replaced programs include `ps`, `login`, `netstat`, `chfn`, `chsh`, `crontab`, `find`, `inetd`, `passwd`, `rshd`, `sysklogd`, and `tcp wrappers`. A password sniffing program is also installed as part of the Linux rootkit version 4, which leads into the data extraction phase.

5.4.2.6 Data Extraction

The intruder installs and runs a password sniffer that takes user names and passwords from telnet and ftp sessions on the LAN.

5.4.2.7 Attack Relay

The intruder installs and runs a distributed denial of service agent, such as Trinoo, TFX, or Stacheldraht. The intruder can then use the system to execute attacks against other networked sites.

5.4.2.8 Derived IDS Requirements

The software fault trees involved in this intrusion helped identify the software requirements for the mobile agent software tasked with detecting the FTP bounce attack. Software requirements on the IDS are to monitor commands and responses in an FTP session, to monitor rsh connections, and to correlate outputs from the two monitors to determine whether an FTP bounce attack was attempted and whether the attack was successful. The analysis does not say anything about how the monitoring should be implemented or performed, but merely yields requirements for the intrusion detection system.
5.4.2.9 Countermeasures for the FTP Bounce Attack

Each of the steps in the intrusions detailed above is a scenario which fits a minimum cut of the corresponding fault tree. Inspecting the minimum cuts for each intrusion leads us to the best point at which countermeasures could be applied. Countermeasures in intrusion detection systems typically include alerts to the system manager (via email, paging, or simply log messages), termination of network connections or logins, and disabling user accounts.

We examined the minimum cut from the penetration tree for the FTP Bounce Attack and informally considered the cost of applying countermeasures at each node. Cost included the complexity of the software required and the effect on the legitimate users of the system. It appears that the lowest cost countermeasure would be killing the TCP connection made from the FTP server to the RSH server; countermeasures at other nodes would either be prohibitive to implement, prevent legitimate uses of the FTP or RSH services, or be too late to terminate the FTP bounce intrusion.

5.4.3 Results of Software Fault Trees for Intrusions

Software fault trees for intrusions explore the necessary and sufficient combinations of events that lead to exploitation of a vulnerability. Development of fault trees for intrusions enables a variety of discovery and verification activities. A brief summary of activities are given below.

5.4.3.1 Requirements Identification & Analysis

Fault trees document properties of intrusions and allow for analysis of intrusion properties.

Determining Requirements Each minimum cut models an intrusion sequence that the software may be required to recognize. Identification of leaf events in the fault tree illustrates what components of a distributed system must be monitored to detect the intrusion. In addition, analysis of intrusion fault trees exposes conditions where countermeasures may be successfully applied by an intrusion detection system to intervene before the intrusion is successful.

Fault Detectability Analysis This refers to the ability of the system to detect the problem if it appears during system operation [30]. Determining which characteristics of intrusions can be monitored is an essential part of the requirements analysis for an IDS. Each leaf event in the fault tree must be analyzed to determine what part of the monitored system provides evidence of the event’s occurrence. If there is no way to obtain evidence of the event, it is undetectable. Marking these events in the fault
tree as undetectable allows analysis of which intrusions would be particularly difficult to detect, and may give hints regarding ways to prevent such intrusions from occurring.

For example, there exist certain intrusive events which do not have any discernible effect in a site's distributed system. Such events include “DNS zone transfers” from off-site secondary name servers and passive password sniffing. In the case of password sniffing, the ways to prevent the intrusion from occurring include using one-time passwords or encrypting terminal sessions to avoid transporting passwords in cleartext.

5.4.3.2 Requirements Evolution & Incremental Development

Prioritization of Requirements The addition of historical information regarding likelihood and severity on the leaf nodes would assist in prioritizing requirements. In addition, based on severity and likelihood information from the fault trees, alert priorities could be encoded in an intrusion detection system.

New Attacks Newly-discovered attacks need to be integrated into the intrusion fault tree. Such new information may encourage re-organization of the fault tree, as when a new attack depends on a set of circumstances that is already diagrammed in the fault tree, or the addition of a new subtree (either new or reused). The changes necessary in the intrusion fault tree to incorporate information about newly-discovered attacks will then guide the necessary modifications of the intrusion detection system requirements and design to detect the new attacks.

5.4.3.3 Verification

Once confidence is established in the software fault tree, primarily through expert review, the design of the intrusion detection system can then be traced to the software fault tree to determine its completeness and correctness.

5.4.4 Testing Intrusion Fault Trees

We would like to use intrusion fault trees to provide requirements for an intrusion detection system. It will be necessary to test the developed fault trees to find and correct faults before creating a design and implementation of an IDS based on the requirements.

Given a subtree of an SFTA that describes related intrusive events, define the subtree to be an equivalence class for the set of intrusions. Select one or more representative minimum cuts of subtree to
be tested. Then, given scenarios which are positive and negative examples of the intrusions, execute the intrusions and determine whether the subtree accurately matches the events. The scenarios form a set of representative test cases for the equivalence class (based on the testing strategy shown by Puketza et al. [110]).

The set of scenarios will be useful to test the design and implementation of an intrusion detection system if the SFTA is used as a requirements specification for an IDS. Because an SFTA is a high-level specification with few constraints and details, it will match all positive use cases and some negative use cases. It will be necessary to augment the use cases with information regarding whether it is expected to be identified by the fault tree as intrusive and whether it is expected to be identified by the design and implementation as intrusive.

5.5 Summary

The use of software fault tree analysis to model intrusions has been presented with supporting examples and illustrative uses. Division of fault trees for intrusions into various stages was examined. Sample fault trees for each of the intrusion stages were described, and two intrusions were examined using the developed fault trees. An example use of fault trees for countermeasures analysis was described.

Software fault tree analysis of intrusions results in a number of benefits. SFTAs enable structured analysis of intrusions, including severity and probability analysis. SFTAs assist the intrusion detection system development process by modeling intrusions, helping identify priorities for development, and specifying requirements for an intrusion detection system. SFTA models of intrusions may assist the verification process for an intrusion detection system and help identify appropriate countermeasures.

We have begun to formalize the use of the developed software fault trees to drive the development of an intrusion detection system design. Software fault tree models of intrusions provide an indirect requirements description for the design of the IDS. The resulting design is based on Colored Petri Nets and implementable in mobile agents, which will be described in the following chapters. SFTA models of intrusions may also assist the verification process by providing test case scenarios (paths of attack) that the IDS is required to detect.
6 COLORED PETRI NET MODELING FOR INTRUSION DETECTION

This chapter examines the use of Colored Petri Nets (CPNs) for modeling intrusion detection systems. CPNs are a compact model often used to describe distributed systems. CPNs closely correspond to system implementations and as such are suited for use as a design tool. When hierarchically composed, CPNs also closely correspond to the design of our agent-based intrusion detection system.

The construction of CPNs to model detection of sample intrusions are presented. The correspondence of CPNs to code in our Java-based agent system is also examined.

6.1 Introduction

A secure computer system provides guarantees regarding the confidentiality, integrity, and availability of its objects (such as data, processes, or services). However, systems generally contain design and implementation flaws that result in security vulnerabilities. An intrusion takes place when an attacker or group of attackers exploit security vulnerabilities and thus violate the confidentiality, integrity, or availability guarantees of a system. Intrusion detection systems (IDSs) detect some set of intrusions and execute — some predetermined action when an intrusion is detected.

Intrusion detection systems use audit information obtained from host systems and networks to determine whether violations of a system's security policy are occurring or have occurred [2]. Our Multi-Agents Intrusion Detection System (MAIDS) [48] uses mobile agents [9] in a distributed system to obtain audit data, correlate events, and discover intrusions.

The MAIDS system comprises (1) stationary data cleaning agents that obtain information from system logs, audit data, and operational statistics and convert the information into a common format, (2) low level agents that monitor and classify ongoing activities, classify events, and pass on their information to mediators, and (3) data mining agents that use machine learning to acquire predictive rules for intrusion detection from system logs and audit data.

However, we find the lack of a sound theoretical model and systematic method for the construction to be an impediment to development of the system. It is difficult to determine exactly what data elements
should be correlated to determine whether an intrusion is taking place on a distributed system. It is also difficult to determine what data was necessary to discover intrusions. Verification of the proper operation of the IDS was possible only informally by executing the IDS and checking its results. A model of intrusion detection is necessary to describe how the data should flow through the system, determine whether the system would be able to detect intrusions, and potentially suggest points at which countermeasures could be implemented.

Software Fault Tree Analysis (SFTA) [68] is used first to model intrusions and develop requirements for the IDS. SFTA is a natural fit as the IDS design resembles a tree where data is obtained at the leaf nodes, travels up through the internal nodes as data is correlated with other information, and rises to the root node when an intrusion is identified.

We do not interpret the Software Fault Trees (SFT) directly as requirements, unlike [45], where the SFT has a formal semantics. A less formal approach was desired in the intrusion application because we want the fault tree to be developed and maintained by system support personnel rather than by experts in formal specification. It is primarily the support personnel’s knowledge of the system and its vulnerabilities that the fault tree is intended to capture.

The SFTs are then mapped into Colored Petri Nets (CPNs) [55] that serve as the design for the IDS. CPNs are a well-documented and frequently-used abstraction for modeling complex and distributed systems. They appear particularly suited for describing the gathering, classification, and correlation activities of an intrusion detection system. CPNs provide a design upon which to base further development of our agent-based IDS. Our work with modeling intrusion detectors as CPNs has shown that CPNs show promise for:

- Providing a formal foundation for the agent-based distributed intrusion detection system; and
- Allowing analysis of the intrusion detection system for:
  - discovering inconsistencies between components of the system,
  - finding ideal places in the monitored system for security improvements, and
  - proving that certain attacks can not be successful if a system is changed to remove or disallow vulnerabilities.

These CPNs are then mapped into the implementation as mobile agents that form the distributed intrusion detection system. We develop CPNs that detect single attacks or attack components, and then compose these CPNs to match intrusions whose attacks proceed through the temporal stages described
in Section 1.3. Composition of the CPNs provides correlation of attacks by unifying the tokens. Such correlation assists identification of the attackers and the source of the attack. Discovering how attacks may fit together into complete intrusions can assist the composition of CPNs. For example, if detectors for single attacks are developed, data mining techniques such as frequent episodes may discover groups of attacks that occur in combination. A detector for the group of attacks can be made by composing the individual attack detectors together.

A distributed implementation of the CPN model using mobile agents can provide a reliable, robust, and efficient intrusion detection system that can highlight intrusions and provide very useful information to the security analyst in the form of a trail of transitions through which CPNs passed.

The SFTA approach applies safety engineering techniques to the intrusion detection domain for developing IDS requirements. Similarly, the CPN model for an intrusion detection system provides a way to develop an IDS design. Finally, the software mobile agent intrusion detection architecture enables development of an efficient, distributed intrusion detection system. However, each part of the development process — SFTs, CPNs, and software agent implementation — is distinct, and we want to ensure the satisfaction of each stage in the development process with respect to the previous stages. An algorithmic transformation is useful to convert requirements into design templates and design into implementation templates. The constructive approach helps ensure the correctness of the design with respect to requirements and correctness of the implementation with respect to the design.

We present the process for developing a CPN design for the IDS using a requirements specification based on a SFTA of intrusions, and we show the procedure for creating an implementation of a distributed agent-based IDS from the CPN design.

The rest of this chapter follows this sequence. Color Petri Nets are explained. Intrusions are divided into temporal stages to enable development of detectors that may be aggregated to form a complete intrusion detection system. Finally, motivating examples of attacks are explained.

6.2 Colored Petri Nets

(CPNs) are a powerful modeling technique for complex systems [55]. CPNs combine state and action into a single diagram through the use of tokens of various colors (colors can be thought of as data types) which reside in places (or states). Tokens move from one place to another through transitions. Transitions allow tokens to pass if all input arcs are enabled (meaning tokens are available for each input arc). Tokens entering from multiple places may be merged (or unified) at transitions. Tokens leaving transitions may be duplicated to multiple destination places. CPNs may be organized
in hierarchical fashion to allow reuse and top-down or bottom-up development, as is possible in modern programming languages.

In a graphical representation of a CPN, places are denoted by ovals or circles, transitions are denoted by squares or rectangles, and lines with arrows denote arcs. If a predicate or tuple is written next to an arc, a token must satisfy the predicate or unify with the tuple before it may pass through the arc. Token colors are defined at the top of each CPN in terms of tuples of standard values, such as strings or integers (tokens may also be defined as data structures). Places may be labeled with a particular color by an italicized label.

Formally: "a non-hierarchical CP-net [Colored Petri Net] is a tuple \( CPN = (\Sigma, P, T, A, N, C, G, E, I) \) satisfying the requirements below [55]:

1. \( \Sigma \) is a finite set of non-empty types, called color sets.
2. \( P \) is a finite set of places.
3. \( T \) is a finite set of transitions.
4. \( A \) is a finite set of arcs such that:
   \[ P \cap T = P \cap A = T \cap A = \emptyset. \]
5. \( N \) is a node function \( A \to P \times T \cup T \times P. \)
6. \( C \) is a color function \( P \to \Sigma. \)
7. \( G \) is a guard function defined from \( T \) into expressions such that:
   \[ \forall t \in T : [\text{Type}(G(t)) = \text{Boolean} \land \text{Type}(\text{Var}(G(t))) \subseteq \Sigma]. \]
8. \( E \) is an arc expression function defined from \( A \) into expressions such that:
   \[ \forall a \in A : [\text{Type}(E(a)) = C(p(a))_{MS} \land \text{Type}(\text{Var}(E(a))) \subseteq \Sigma] \]
   where \( p(a) \) is the place of \( N(a). \)
9. \( I \) is an initialization function defined from \( P \) into closed expressions such that
   \[ \forall p \in P : [\text{Type}(I(p)) = C(p)_{MS}]." \]

A hierarchical CPN is a set of CPNs composed into a hierarchy. The two building blocks of hierarchical CPNs are substitution transitions and fusion places. Substitution transitions and fusion places allow the construction of a hierarchical CPN by combining a number of non-hierarchical CPNs. A hierarchical
CPNs may be translated into a behaviorally equivalent non-hierarchical CPN, and vice versa. Hierarchical CPNs are important to our design of the IDS because they allow construction of detectors for attacks that can be composed into detectors for complete intrusions.

CPNs have been applied to a variety of problem domains, including security, network protocols, mutual exclusion algorithms, VLSI chip designs, and chemical manufacturing systems [56]. Petri Nets have also been applied to the safety domain [71], which is closely related to the security domain [116].

We use Riuu's stages [115] to structure the analysis of the intrusions discussed in this chapter. The CPNs examined in this chapter generally correspond to the reconnaissance, vulnerability identification, and penetration phases of intrusions, as these first three stages are essential to intrusions. The previous chapter examined Software Fault Tree Analysis, which assist generation of CPNs for other attacks and other stages of intrusions.

### 6.2.1 Descriptions of Examined Intrusions

One type intrusion we consider in detail is an FTP bounce attack. The "FTP Host" provides an anonymous FTP service that allows uploads and the "Target Host" provides a remote shell service that trusts the users on the "FTP Host."

1. In preparation, the attacker creates a file containing a valid remote shell (rsh) message such as
   
   ```bash
   \0root\0root\0xterm -display bad.hacker.org:0.0
   ```
   
   which means "I am the user root on the local computer. I wish to execute a command on the remote computer as the user root, and the command I wish to execute will open a terminal window from the remote computer on my screen."

2. The attacker scans for valid hosts in the target's network. For the purposes of our spatially distributed attack, assume the attacker discovers at least two host systems in the target's networks.

3. The attacker scans for listening TCP ports on the target network's valid hosts. Assume the attacker discovers a vulnerable anonymous FTP server listening at TCP port 21 on the "FTP Host", and a remote shell daemon (rshd) listening at port 514 on the "Target Host."

4. The attacker uploads the previously created file to the anonymous FTP server on the "FTP Host."

5. The attacker uses a "feature" of the FTP protocol to tell the FTP server to send the next download to port 514 on the "Target Host". Then the attacker issues a command to the FTP server that initiates a "download" of the file containing the rsh message. If the "Target Host" trusts the
users on the “FTP Host”, the remote shell daemon on the “Target Host” accepts the message and executes it due to an authentication vulnerability in the remote shell protocol.

6. The “Target Host” opens a terminal window on the attacker’s X Window server that provides the attacker with root privileged shell. The attacker may proceed with any number of activities, including: changing passwords or adding users; reading or changing any file on the system; erase traces of his/her presence; and install tools to sniff passwords, provide backdoors for future access, and disguise his/her activities.

We also consider an intrusion based on an NFS file handle guessing attack.

1. The attacker scans for valid hosts in the target’s network. To demonstrate the spatially distributed attack, assume the attacker discovers at least two host systems in the target’s networks.

2. The attacker scans for available UDP services on the target network’s valid hosts. Assume the attacker discovers the NFS file service operating at port 2049 on the “NFS Host.” The NFS file service typically does not require authentication and will provide a way for the attacker to modify files on the “NFS Host.”

3. The attacker scans for listening TCP ports on the target network’s valid hosts. Assume the attacker discovers a remote login daemon (rlogind) listening at port 513 on the “Target Host.”

4. The attacker uses an NFS exploit tool (perhaps based on nfsbug.c [130]) to discover an NFS file handle for the root of the file system containing the file /etc/hosts.equiv.

5. The attacker uses the learned root NFS file handle to obtain the NFS file handle for /etc/hosts.equiv.

6. The attacker uses an NFS write remote procedure call on /etc/hosts.equiv’s file handle to add the attacker’s host name to the file.

7. Assuming the “Target Host” uses /etc/hosts.equiv from the “NFS Host,” the attacker may now use rlogin to remotely login from his host to “Target Host” as the user root. The attacker will be able to login without a password because the remote login server on the “Target Host” now mistakenly trusts the users on the attacker’s host.

8. The attacker may proceed with any number of activities, including: changing passwords or adding users; reading or changing any file on the system; erase traces of his/her presence; and install tools to sniff passwords, provide backdoors for future access, and disguise his/her activities.
6.2.2 CPN Models of Intrusion Detection Systems

Detectors for the intrusions explained in Section 6.2.1 were modeled using the hierarchical Colored Petri Net notation described by Jensen [55].

6.2.2.1 FTP Bounce Attack Detector

Figures 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, and 6.8 collectively represent a hierarchical CPN for the FTP Bounce intrusion.

Figure 6.1 CPN for the entire FTP Bounce intrusion (ftp-bounce-hier-cpn)

Figure 6.1 shows the temporal sequence of the attack at the highest level of representation and provides an example of composition of CPNs. It illustrates the combination of the reconnaissance, vulnerability identification, and penetration phases. Tokens are expected to always be available at the Normal place and advance through the FTP Bounce Scan transition when scanning activities occurs. Tokens are taken from Recon place when a connection is made to a potentially vulnerable FTP server. Tokens finally move through the FTP Bounce Attack transition when an attack occurs. The boxed "HS" next to the FTP Bounce Scan, Vulnerability ID, and FTP Bounce Attack transitions denotes "hierarchy and substitution" as described by Jensen’s hierarchical CPN diagrams [55]. The three lines in the dashed box to the left of the "HS" give the name of the CPN to substitute, the outgoing place to substitute, and the incoming place to substitute. For the FTP Bounce Scan transition, the "ftp-bounce-scan" CPN is substituted (Figure 6.2), its outgoing place named "Recon Successful" is substituted for
Figure 6.2 CPN for the reconnaissance stage of the FTP Bounce intrusion (ftp-bounce-scan-cpn)

Figure 6.3 CPN for scanning for a host providing the FTP service (ftp-bounce-scan-ftp-cpn)
Figure 6.4 CPN for scanning for a host providing the RSH service (ftp-bounce-scan-rsh-cpn)

Figure 6.5 CPN for host scanning detector (host-scan-cpn)
color IP_QUAD = product unsigned long *
             unsigned short * unsigned long * unsigned short;
color PORT_SCAN = IP_QUAD;

Figure 6.6 CPN for port scanning detector (port-scan-cpn)

Figure 6.7 Simple CPN for identifying FTP server vulnerable to FTP Bounce (ftp-bounce-vuln-cpn)
Figure 6.8 CPN for the penetration stage of the FTP Bounce intrusion (ftp-bounce-cpn)
this CPNs place named "Recon", and its incoming place named "Incoming" is substituted for this CPNs place named "Normal".

Figure 6.2 models detection of the reconnaissance phase of the intrusion. It watches for the attackers actions that would be used to discover the presence of one or more hosts that provide FTP and RSH services. Unlike the parent CPN of figure 6.1 which is serial, this CPN allows discovery of "FTP Scanning" and "RSH Scanning" events in parallel. When both the FTP Scanning and the RSH Scanning event from the same source host are noticed, the tokens are merged and moved to the "Recon Successful" place. The "FTP Scanning" transition is substituted with the CPN in Figure 6.3. and the "RSH Scanning" transition is substituted with the CPN in Figure 6.4.

Figures 6.3 and 6.4 represent the process of detecting the scans for the host(s) that offer FTP and RSH services, respectively. Both these CPNs substitute the lowest-level CPNs, shown in Figures 6.5 and 6.6, for host scanning and port scanning detection. Watching for host scanning is performed in parallel with watching for port scanning, and tokens are unified when they have matching source and destination IP addresses.

Figure 6.5 is the non-hierarchical CPN that outputs tokens when a host is discovered by scanning. Hosts may be actively discovered through the use of ICMP Echo Requests to a network's broadcast address, unicast ICMP Echo Requests to individual addresses, TCP connection requests to well-known ports, or TCP "stealth" scanning packets. Any of these scanning techniques will betray the IP address used by the attacker. If the IP address used in scanning matches the IP address used in other parts of an intrusion, stronger identification of the source of the intrusion can be made by correlating the events based on the IP address.

Figure 6.6 is the non-hierarchical CPN that outputs tokens when TCP services are discovered on a host by scanning. Available services on a host may be actively discovered by TCP SYN (connection establishment) segments or TCP segments with some combination of the flags FIN, URG, PSH, ACK, RST set (also known as "stealth" scanning segments). As a matter of implementation, tools such as SNORT or Portsentry may be used to detect scans and provide data that can be turned into tokens.

Figure 6.7 shows a simple CPN for the vulnerability identification phase. It allows tokens through when a connection is made to a possibly vulnerable FTP server. Because of the vast number of FTP servers available, a check against a few known invulnerable FTP servers is made as part of the "Vulnerable?" transition: if the check fails, the token is allowed to pass under the assumption that the FTP server may be vulnerable.

Figure 6.8 show a model that detects the actions taken by the attacker to exploit vulnerable FTP
and RSH services. The existence of a place ("Network Monitor: FTP") that generates tokens from observed FTP commands and responses is assumed: such a place could sniff network traffic or obtain events directly from a cooperative FTP server. This figure shows how the attacker’s first command is detected (using regular expressions for matching), the FTP server’s “command successful” response is detected, and finally the attacker’s second command, which completes the attack, is detected.

6.2.2.2 NFS Misuse Detector

Figures 6.9, 6.10, 6.11, 6.12, and 6.13 show a hierarchical CPN for the NFS filehandle guessing intrusion. Many similarities and commonalities exist between this attack detector CPN and the FTP Bounce detector CPN, including the host scanning detector (Figure 6.5) and the port scanning detector (Figure 6.6) which are completely shared.

```
color IP_QUAD = product unsigned long * unsigned short * unsigned long * unsigned short;
color NFS = product string * string * IP_QUAD;
color NFS_ATTACK = product DUAL_SCAN * NFS;
color PCRT_SCAN = IP_QUAD;
color DUAL_SCAN = product PCRT_SCAN * PCRT_SCAN;
```

![CPN Diagram]

Figure 6.9 CPN for the entire NFS FileHandle Guessing intrusion (nfs-filehandle-hier-cpn)

Figure 6.9 shows the attack detector at the highest level of representation. In the reconnaissance phase of the attack, it is expected that the attacker will scan the organization’s network to discover the hosts that provide NFS and rlogin services. The availability of these services also satisfies the vulnerability identification phase, as NFS is itself vulnerable and it is not possible to test rlogin for vulnerabilities short of attempting to exploit it. In the penetration phase of the attack, the attacker is expected to exploit the vulnerabilities in NFS and rlogin to gain unauthorized access to the systems.
Figure 6.10 CPN for the reconnaissance stage of the NFS Filehandle Guessing intrusion (nfs-filehandle-scan-cpn)

Figure 6.11 CPN for scanning for a host providing the NFS service (scan-nfs-cpn)
Figure 6.12 CPN for scanning for a host providing the Rlogin service (scan-rlogin-cpn)

color IP_QUAD = product unsigned long * unsigned short * unsigned long * unsigned short;
color NFS = product string * string * IP_QUAD;
var ip_quad : IP_QUAD;
var src_host, dest_host : unsigned long;
var src_port : unsigned short;

Figure 6.13 CPN for the penetration stage of the NFS Filehandle Guessing intrusion (nfs-filehandle-cpn)
Figure 6.10 is substituted for the “NFS & Rlogin Scan” transition, and Figure 6.13 is substituted for the “NFS File handle Attack” transition.

Figure 6.10 models detection of the reconnaissance phase of the attack. It watches for attacker activity that indicates the attacker is discovering one or more hosts that provide NFS and rlogin services. This CPN allows identification of NFS scanning events in parallel with rlogin scanning events. When corresponding NFS and rlogin scanning events have occurred, the merged token moves to the “Scan Successful” place. The “NFS Scanning” transition is substituted with the CPX of Figure 6.11, and the “Rlogin Scanning” transition is substituted with the CPX of Figure 6.12.

Figures 6.11 and 6.12 represent the host and port scan detection for NFS and rlogin services. As with the FTP Bounce scan CPNs, both these CPNs substitute the lowest-level CPNs, shown in Figures 6.5 and 6.6, for host scanning and port scanning detection. Watching for host scanning is performed in parallel with watching for port scanning, and tokens are unified when they have matching source and destination IP addresses.

Figure 6.13 models detection of the attacker’s actions which exploit vulnerable NFS and rlogin services. The detector watches for NFS `getattr` requests from an unexpected client, followed by an “OK” response from the NFS server. These steps correlate loosely with the methods used in van Doorn’s `nfsbug` program [130], and could be expected as an attacker attempts to find an NFS file handle for a directory on an NFS-exported file system. If an attacker is able successfully to discover or guess an NFS file handle for a file system containing the `/etc/hosts.equiv` file or any user’s `.rhosts` files, the attacker will then likely be able to find the NFS file handle for one of these files and issue an NFS `write` to the file and thus give himself login privileges. The attacker can use these newly-gained privileges to login to the target host.

### 6.2.2.3 Anomaly Detection Rules as a CPN

The FTP Bounce and NFS Misuse CPX models misuse intrusion detection. To illustrate the completeness of the CPN approach for intrusion detection, rules learned by a data mining algorithm for anomaly detection are modeled with CPNs. In this case, rules that detect sendmail anomalies [48] were modeled. Figure 6.14 shows the rule set from RIPPER. Each line consists of a rule where the left side is a classification to be assigned if all of the terms on the right side are true. Rules are tested in order. Figure 6.15 shows an equivalent CPN for the rules. Constants in the tuples on the edges between transitions and the “Classified” place set the classification and the number of correct & incorrect matches from the training set.
good : - aayk=f, aacp=t (170/0).
good : - accc=t, aaaa=t (152/0).
good : - aai=t (161/36).
good : - aaz=f, acn=t (6/0).
good : - acb=t (2/0).
default bad (480/19).

Figure 6.14 Rules for Sendmail anomaly detection

Figure 6.15 RIPPER rules for Sendmail modeled as a CPN
The algorithm for translating serially executed rules to a CPN is as follows:

1. For each non-default rule \( r \) in the rule set:
   
   (a) Create one transition with an incoming edge whose predicate evaluates true when the rule \( r \) is satisfied. Connect the outgoing edge to the final place and set the classification & confidence measures for rule \( r \) on the token via this edge.
   
   (b) Create one transition with an incoming edge whose predicate evaluates true when the rule \( r \) is not satisfied. Connect the outgoing edge to an intermediate place that provides input to the immediately-following rule.

2. For the default rule \( d \), create an incoming edge from the previous rule's intermediate place. Connect the outgoing edge to the final place and set the classification & confidence measures for rule \( d \) on the token via this edge.

Two transitions are created in the CPN for each rule to duplicate the sequence nature of the rules. The first transition represents matching a particular rule, and the second transition represents a mismatch for the particular rule. Together, the predicates on the arcs into the two transitions form a tautology, and thus any token of the correct color will take one of the two paths. Tokens then move through "mismatch" transitions until they match a rule, after which the token is extended with the assigned classification and associated confidence attributes.

### 6.3 Translating CPNs to Code

An algorithmic translation is presented that generates code from CPNs. A developer creates an implementation by following the algorithm to write the code. The developed code contains places that hold tokens and transitions that match and unify tokens.

By using the translation algorithm to convert CPN designs to code, we can be certain that the code implements the CPN design. Also, if the translation algorithm preserves CPN semantics, any analysis performed on the CPN design should also apply to the implementation. Finally, creating an implementation in code allows the developer to improve performance over the execution of a general CPN.
6.3.1 Java-based CPN implementation algorithm

The algorithm for translating a CPN to an implementation in Java is as follows (the nomenclature is based on the object oriented programming model used in the Java programming language):

1. For each place in a CPN, instantiate a Place object that implements the IPlace interface:

2. For each transition in a CPN, create and instantiate a Transition class that implements the ITransition interface such that the run() method takes and unifies tokens from incoming places and puts output tokens in outgoing places as specified by the arcs and guards in the CPN.

The algorithm preserves the CPN semantics in the implementation and allows for efficient execution. Performance may be increased over the execution of a general CPN by optimizing the code segments to fit the intrusion detection application. For example, matching and unifying tokens is computationally intensive, at least $O(n^2)$, in the general case. Matching and unifying tokens in specific cases may be made much faster by knowing in advance what types of tokens are expected and using that knowledge to implement faster algorithms. For example, an implementation could match tokens in $O(n \log n)$ time based on binary search or $O(n)$ time based on hash tables.

The Token class is implicitly defined as a data type that holds the token data. The Place class is defined to implement storage areas for tokens and provide methods to receive tokens from and give tokens to Transitions. A sample interface implemented by a Place is shown in Figure 6.16. The associated interface for token matching is shown in Figure 6.17, which allows a transition to selectively accept tokens that satisfy the predicate. Implementations of the TokenMatcher interface may make use of computationally efficient algorithms, such as binary searches or hash table lookups, to match tokens.

```java
public interface IPlace
{
    /** Obtain tokens that satisfy the ITokenMatcher. */
    public Token[] getTokens(ITokenMatcher tm);
    /** Store tokens. */
    public void storeToken(Token t);
}
```

Figure 6.16 Sample Java interface for a Place
public interface ITokenMatcher
{
    /** Predicate that returns true for matching tokens. */
    public boolean tokenMatches(Token t);
}

Figure 6.17 Sample Java interface for a Token Matcher

The Transition class is defined which provides methods that obtain tokens from incoming Places, operate on tokens when unifying tokens are received from all incoming Places, and send tokens to outgoing Places. Figure 6.18 shows a simple interface that could be implemented by a Transition.

public interface ITransition
{
    /** Perform some operation (e.g., unification). */
    protected Token[] operateOnTokens(Token[] inputTokens);
    /** Execute the transition: obtain tokens from incoming places, operate on them, and store them in outgoing places. */
    public void run();
}

Figure 6.18 Sample Java interface for a Transition

Each transition implementation must connect to its incoming and outgoing Places and implement token matching and unification methods, operations on tokens, and token output methods. Custom tailored, efficient implementations of each of these methods may improve performance over that of directly executing CPNs.

6.3.2 Preservation of CPN Semantics

The implementation of the CPNs as classes in Java satisfy the following properties of the CPNs. By examination, the properties of CPNs from Section 6.2 are satisfied:

1. Each token type is a color in set $\Sigma$.
2. Each instance of a Place is an element in set $P$.
3. Each instance of a Transition is an element in set $T$.
4. Arcs $A$ between Places and Transitions are encoded in the Transitions; they are finite in number and satisfy requirement 4 as they are distinct from Places and Transitions.
5. The encoding of Arcs in Transitions is the node function $N$.

6. The encoding of outgoing Arcs in Transitions determines the colors of tokens that enter Places, giving the color function $C$.

7. The predicate in interface ITokenMatcher implements the guard expressions in $G$ for each transition in $T$.

8. The predicate in interface ITokenMatcher also implements the arc expressions in $E$ for each arc in $A$.

9. Each instance of a Place has initial state according to the initialization function $I$.

The implementation of transitions and places may impose additional constraints not present in the CPNs in order to obtain efficiencies for particular expected token colors. As long as only expected token colors exist in the places, the CPN semantics should be satisfied by the implementation.

6.3.3 Testing CPNs and Implementation

A set of use cases (positive and negative examples of intrusions) will be developed to test the intrusion detection system requirements. The CPN design will be tested using the use cases to observe the behavior of the CPN and verify correct functionality. Equivalence classes may be used to test representative samples from groups of intrusions to reduce the testing effort [110].

Since the requirements model is less detailed than the CPN and may not be as expressive as a CPN model, the CPN design further constrains the sets of events that will be identified as intrusions. Thus, some use cases that are identified by the requirements as intrusions will not be considered intrusions by the CPN model and the intrusion detection system implementation. Each use case must be annotated to describe whether the requirements and/or design will identify the use case as an intrusion.

Four ways of analyzing CPNs include:

1. Interactive simulation - Execute a CPN model in a way similar to interactively debugging a program.

2. Automatic simulation - Investigate functional correctness and performance of a CPN model by executing a CPN at full speed.

3. Creating occurrence graphs - Determine reachability of nodes in a CPN model.
4. Place invariants - Prove user-specified predicates to be satisfied for all reachable system states to prove properties such as absence of deadlock.

Place invariants in particular may be useful for the intrusion detection CPN design, as they may allow invariants to be derived from requirements and verified in the CPN design. For example, a place in an FTP bounce attack detector of Figure 6.8 may have an "FTP RESPONSE" token only if there exists a matching "FTP COMMAND" token in the CPN, since a command must be issued to receive a response.

Interactive simulation has been performed by building CPNs and simulating their execution in the Design CPN tool [28] using positive and negative examples of intrusions. Automatic simulation has been performed indirectly by building an implementation of CPNs in Java and executing it.

6.4 Summary

The use of formal models for IDSs based on CPNs was examined. CPNs are useful for modeling complex and distributed systems. CPNs provide a convenient model for describing the gathering, classification, and correlation activities of an intrusion detection system. CPNs provide a design which enables development of our agent-based IDS.

This paper details the procedure by which a distributed, agent-based IDS was implemented from a CPN design. Algorithmic approaches are used to create CPN templates from augmented SFTs and agent implementations from CPN designs. The end result is an intrusion detection system which detects the intrusions which were specified by the original requirements.

Dividing components of intrusions into temporal stages allows the development of CPNs that detect individual attacks. Composition of the CPNs into a hierarchy models the correlation of individual attacks to detect complete intrusions.

An algorithm is used to convert CPNs into agent implementation templates. The implementation preserves the properties of the CPN design while providing agents for use as a distributed intrusion detection system.

Agents in our prototype intrusion detection system function as CPN places and transitions. Places are generally static agents which either act as a source of information or hold information until a transition requests it. Transition agents are the active components which accept tokens from places, act on or unify the information in the tokens, and pass the resulting tokens to other places. Viewing MAIDS agents and data as an implementation of a Colored Petri Net has conveniently generalized the system and enabled further development.
We are implementing an FTP bounce attack detector and an NFS attack detector in MAIDS based on the CPNs detailed in this chapter. Thus far, we have concentrated mainly on the penetration phase of intrusions, but have also begun development of CPNs for other stages of intrusions.
7 SYSTEM INTEGRATION

The requirements specifications using Software Fault Tree Analysis (SFTA) [68], the design specifications using Colored Petri Nets, and the software mobile agent implementation of an intrusion detection system are closely coupled together. An algorithmic approach for creating CPN templates from SFTs is proposed to ensure satisfaction of the SFT-based requirements by the CPN design. Likewise, an algorithmic approach to creating an agent-based implementation from the CPN design is demonstrated.

7.1 Introduction

The Software Fault Tree for Intrusions approach is a method for applying safety engineering techniques to the intrusion detection domain for developing IDS requirements. Similarly, the Colored Petri Net model for an intrusion detection system provides a way to develop an IDS design. Finally, the software mobile agent intrusion detection architecture enables development of an efficient, distributed intrusion detection system. However, each component — SFTs, CPNs, and software agent implementation — is separate from the others, and we want to ensure the satisfaction of each stage in the development process with respect to the previous stages.

An algorithmic transformation is useful to convert requirements into design templates and design into implementation templates. The constructive approach helps ensure the correctness of the design with respect to requirements and correctness of the implementation with respect to the design. Figure 7.1 illustrates the development process.

7.2 From SFT Requirements to CPN Design

Two main issues are considered to enable developing CPN designs from SFT requirements. First, SFTs are augmented with constraints to more accurately describe intrusions. Constraints include temporal ordering, trust relationships, and context. Second, an algorithmic approach to converting SFTs with constraints to CPN templates is examined.
Fault Tree Analysis

Correctness-preserving restriction

Augmented Fault Tree Analysis

Model of intrusions
* Captures requirements for detection

Correctness-preserving mapping

Colored Petri Nets

System-specific model of intrusions
* Eliminates false positives

Correctness-preserving translation

Template Classes & Agents

System-specific model of intrusion detection
* Hierarchical
* Composable
* Design Level
* Simulation and analysis
* Template for coding

Mobile Agents
* Distributed
* Implementation
* Prototyped

Future

Testing

Verification
* FTA functional testcases
* CPN structural testcases
* Experimental deployment

Present

Figure 7.1  IDS development process
7.2.1 SFTA for Intrusion Detection Systems

We do not interpret the SFT directly as requirements, unlike [45], where the SFT has a formal semantics. A less formal approach was desired in the intrusion application because we want the fault tree to be developed and maintained by system support personnel rather than by experts in formal specification. It is primarily the support personnel's knowledge of the system and its vulnerabilities that the fault tree is intended to capture. To understand this, a brief description of the larger IDS system is in order.

The intrusion SFT described here is the requirements phase of a larger effort to provide a more formal framework for building IDS. The IDS will use mobile agents in a distributed system to collect audit data, classify it, correlate information from the different mobile agents, and detect intrusions. The intrusion SFT drives the requirements for these mobile agents and the intrusion detection system. The SFT is mapped, by a correctness-preserving transformation, into colored Petri nets (CPNs) that serve as the design specification of the mobile agents in the IDS. Interactive simulation of these CPNs gives additional verification that the design satisfies the requirements (i.e., blocks the relevant path(s) in the intrusion fault tree). Code for the IDS mobile agents is generated from the CPNs and tested using, among other scenarios, the minimum cuts through the intrusion fault tree. Currently, prototypes exist of each of these phases (i.e., CPNs and mobile agents for a selected set of intrusions) with work on-going to partially automate the code generation.

Two interesting aspects of the requirements phase of this prototype are as follows. First, the intrusion SFT have been interpreted as specifications of the combinations of events that must be detected. That is, the IDS requirements are that each of the intrusion sequences possible in the SFT should be detected as soon (low in the tree) as possible. The leaf events describe what components of a distributed system must be monitored by the mobile agent software. The interpretation of the SFTA serves as the requirements specification.

Second, the intrusion SFT have had to be extended with additional information specific to a particular system prior to their mapping into CPNs. This information is of three types: Trust (which members of a distributed system are trusted by other members); Context (which events must all involve the same host(s) or connection(s), process(es) or session(s)); and Temporal orderings (e.g., which events must be adjacent with no intervening events, or follow within a specific interval of time). Without this additional system-specific information, the IDS yields many false positives, detecting intrusions where, in a specific network, there is none. That is, the set of events marked as intrusions by the SFT is a superset of the set of events that are actually intrusions in any specific network and must be constrained...
by additional network-specific knowledge.

### 7.2.2 Augmented SFT

Constraint nodes are proposed for addition to Software Fault Trees for intrusion modeling. The nodes capture trust, order, and contextual relationships needed to develop a satisfactory model of the intrusions that have been analyzed. With the addition of constraint nodes, the design of an intrusion detection system much more closely corresponds to the requirements for the intrusion detection system specified via an SFT.

The effect of adding constraint nodes may be demonstrated by considering the set $E$ of all combinations of events that make the root node of a plain (unenhanced) SFT "true". The set $I \subseteq E$ of combinations of events that are actual intrusions must also make the root node of the augmented SFT "true". ($|I|$ ought to be much smaller than $|E|$.) The constraint nodes added to an augmented SFT should exclude the vast majority of the combinations of "false positive" events $E - I$. Thus the augmented SFT, enhanced with the constraint nodes described here, will more closely model the requirements for an intrusion detection system.

#### 7.2.2.1 Trust

Members of a distributed system trust other members of the system. For example, a Network File System (NFS) server using AUTH_UNIX authentication trusts the user IDs in client requests. This allows a user on a client host to access files on the file server without having to login to the server.

Explicitly stating a trust relationship that is required for an intrusion to succeed provides information to an intrusion detection system developer that will help derive an accurate matching model for the intrusion.

**Syntax:** $\text{Trusts}(\langle \text{destination} \rangle, \langle \text{source} \rangle)$

where:

- $\text{destination}$: Ordered list of constants and variables describing the trusting destination, such as name of destination host, network, or netgroup and application
- $\text{source}$: Ordered list of constants and variables describing the trusted source, such as name of source host, network, or netgroup and application

**Semantics:** The $\text{Trusts}$ predicate is true if the $\text{destination}$ assigns some trust to the $\text{source}$. Specifying trust relationships in this way allows matching relationships to be unified [117] with other trust
relationships. A trust relationship is true if one of a system’s trust relationships successfully unify with the relationship specified by the Trusts predicate.

**Example:** Such a trust relationship may be:

\[
\text{Trusts}((\text{Rshd}. \text{targetHost}).(\text{sourceHost}))
\]

which states that the remote shell daemon (Rshd) on a targetHost trusts a sourceHost. By convention, elements beginning with upper-case letters are constants, and elements beginning with lower-case letters are variables.

### 7.2.2.2 Context

Certain combinations of intrusive events must occur in some common context. For example, a series of FTP commands and responses need to be grouped by a common network connection to an FTP server.

**Network**  
Network-related events may be related by events involving a single host, a pair of hosts, or a single virtual network connection.

**Single Host**  
A single host that must be a common source or target for network events may be specified as a common context for intrusive events.

**Syntax:**  
\[
\text{Context}((: \text{host Hostname}) (\text{FTNodeList}))
\]

where:

- **Hostname:** Constant name or address of a host or group of hosts, or a variable
- **FTNodeList:** List of one or more fault tree nodes to be included in the context

**Semantics:** The Context predicate is true when the host identified by Hostname is involved in each node specified by the FTNodeList.

**Example:** The context for a single host hostname that may be the target of two simultaneous attacks (indicated by fault tree nodes Attack1 and Attack2) would be  
\[
\text{Context}((: \text{host hostname}) (\text{Attack1, Attack2}))
\]
**Host Pair**  A pair of hosts that must be the source and target for network events may be specified as a common context for intrusive events.

**Syntax:** \( \text{Context}((:hosts \ Hostname_1 \ Hostname_2) (FTNodeList)) \)

where:

- **Hostname_1, Hostname_2:** Constant name or address of a host or group of hosts, or a variable
- **FTNodeList:** list of one or more fault tree nodes to be included in the context

**Semantics:** The \( \text{Context} \) predicate is true when hosts identified by \( \text{Hostname}_1 \) and \( \text{Hostname}_2 \) are involved in each node specified by the \( \text{FTNodeList} \).

**Example:** After host \( \text{attacker} \) performs some reconnaissance against the host \( \text{target} \), the attacker may be expected to perform vulnerability identification against the target. Such a context may be expressed as \( \text{Context}((:hosts \text{attacker} \text{target}) (\text{Reconnaissance}, \text{VulnerabilityID})) \).

**Network Connection**  A pair of hosts communicating using a virtual network connection that must be the source and target for network events may be specified as a common context for intrusive events.

**Syntax:** \( \text{Context}((:conn \ Hostname_1 \ Port_1 \ Hostname_2 \ Port_2) (FTNodeList)) \)

where:

- **Hostname_1, Hostname_2:** Constant name or address of a host or group of hosts, or a variable
- **Port_1, Port_2:** Constant name or number of a network port, or a variable
- **FTNodeList:** List of one or more fault tree nodes to be included in the context

**Semantics:** The \( \text{Context} \) predicate is true when a network connection involving the endpoints identified by \( \text{Port}_1 \) on \( \text{Hostname}_1 \) and \( \text{Port}_2 \) on \( \text{Hostname}_2 \) are involved in each node specified by the \( \text{FTNodeList} \).

**Example:** The commands and responses involved in an FTP upload from a source host to a destination host may be set in a common context by the predicate:

\[
\text{Context}((:conn \text{source ephemeralPort destination 21}) (FTP\_PORT, FTP\_PORT\_OK, FTP\_STOR, FTP\_STOR\_OK))
\]
Host  A common host-based context, such as processes or user sessions, may be necessary for intrusions.

User Session An authenticated user session on a host, such as via telnet, ssh, or ftp, may be a context for related events.

Syntax: \texttt{Context(\{\texttt{user} User Application LHost RHost Terminal LoginTime\} \{FTNodeList\})}

where:

\texttt{User: } Constant name or group of user(s), or a variable
\texttt{Application: } Constant name of method of access (e.g., telnet, ftp, etc.), or a variable
\texttt{LHost: } Constant name of the host to which the user is connected, or a variable
\texttt{RHost: } Constant name of a remote host or group of hosts, or a variable
\texttt{Terminal: } Constant name of a terminal used for access (e.g., tty01), or a variable
\texttt{LoginTime: } Constant specification of time of login, or a variable
\texttt{FTNodeList: } List of one or more fault tree nodes to be included in the context

Semantics: A user session is established using the \texttt{Application} into the \texttt{LHost} from the \texttt{RHost} to the \texttt{Terminal} beginning at time \texttt{LoginTime}. The context involves each node specified by the \texttt{FTNodeList}.

Example: The nodes corresponding to the use of the \texttt{ps}, \texttt{who}, and \texttt{logout} command executed by a user during a login session on the host \texttt{durango} from host \texttt{nova} via telnet may be set in a common context with:

\texttt{Context(\{\texttt{user} username Telnet Durango Nova tty01 startTime\} \{Ps. Who. Logout\})}

Process  Events corresponding to a process (an instance of a program in execution) may be a context for related events.

Syntax: \texttt{Context(\{\texttt{process} ProcessID Program User Host StartTime\} \{FTNodeList\})}

where:

\texttt{ProcessID: } Constant or variable indicating the identification number of the process
\texttt{Program: } Constant or variable indicating the name of the program executed
User: Constant user name, or a variable, of the user who executed the process

Host: Constant or variable name of the host on which the process executed

StartTime: Constant or variable indicating the time the process began executing

FTNodeList: List of one or more fault tree nodes to be included in the context

Semantics: A process executes Program with ID ProcessID with the permissions of User on the Host beginning at time StartTime. The context involves each node specified by the FTNodeList.

Example: If a process opens and writes the system password file, a common context for these events may be:

\[
\text{Context}(\text{process pid program username host startTime}) \ \text{(OPEN \_PASSWD.WRITe \_PASSWD))}
\]

7.2.2.3 Temporal Ordering and Intervals

Events and conditions involved in an intrusion often must occur in a particular order. Explicitly specifying the event ordering excludes other non-intrusive permutations of events from being considered as intrusive. We use Allen and Ferguson's interval temporal logic [1] to develop predicates.

Occurs After An event which takes place must make its node in the fault tree true as long as the existence of that event may be combined with other events to make a parent node true. It seems an event's period may last as long as the context exists in which it may be evaluated. In this sense, "occurs after" is concerned only with the relative start of event's periods.

"Occurs after" is the condition where one event's period is required to start after another event's period has started. The \( \text{Starts}(i, j) \) primitive is true when periods \( i \) and \( j \) begin simultaneously. The \( \text{Meets}(i, j) \) primitive is true when period \( i \) ends adjacent to the time where period \( j \) begins. Let \( \text{Period}(x) \) be the period that node \( x \) is true. Let:

\[
\text{OccursAfter} : \text{Node, Node} \to \text{Boolean}
\]

\[
\text{OccursAfter}(i, j) \equiv \exists m \text{Starts}(\text{Period}(i), m) \land \text{Meets}(m, \text{Period}(j))
\]

Syntax: \( \text{OccursAfter}(\text{FTnode}1, \text{FTnode}2) \)

where:

\( \text{FTnode}1, \text{FTnode}2: \) names of event or boolean expression nodes
Semantics: The event or boolean expression indicated by the node $FTnode1$ becomes true in the time prior to the time that the event or boolean expression indicated by the node $FTnode2$ becomes true.

Example: A response must occur after a command, not before. Such a constraint may be specified by the predicate $OccursAfter(COMMAND, RESPONSE)$.

Adjacent Events  Certain situations exist where an event must occur after another event within the same context with no intervening events. Let:

$$ImmediatelyAfter : Node \rightarrow Node \rightarrow Boolean$$

$$ImmediatelyAfter(i, j) \equiv \neg(\exists n \text{OccursAfter}(i, n) \land \text{OccursAfter}(n, j))$$

Syntax: $ImmediatelyAfter(FTnode1, FTnode2)$

where:

$FTnode1, FTnode2$: names of event or boolean expression nodes

Semantics: The event or boolean expression indicated by the node $FTnode1$ becomes true in the time prior to the time that the event or boolean expression indicated by the node $FTnode2$ becomes true. No intervening events become true between $FTnode1$ and $FTnode2$.

Example: In an FTP session, a command must be followed by a response before a second command is issued. Such a constraint may be specified by the predicate:

$$ImmediatelyAfter(COMMAND1, RESPONSE) \land$$
$$ImmediatelyAfter(RESPONSE, COMMAND2)$$

Interval  An event may be required to follow another event within some amount of time. Let $StartOf(i)$ be the start of discrete time period $i$. The $Overlaps(i, j)$ primitive is true when period $i$ overlaps period $j$.

$$Interval : Node \rightarrow Node \rightarrow \mathbb{R} \rightarrow Boolean$$

$$Interval(i, j, t) \equiv \text{OccursAfter}(i, j) \land \text{Overlaps}((\text{StartOf}(\text{Period}(i)), \text{StartOf}(\text{Period}(i)) + t), j)$$

Syntax: $Interval(FTnode1, FTnode2, t)$

where:

$FTnode1, FTnode2$: names of event or boolean expression nodes

$t$: amount of time after $FTnode1$ becomes true in which $FTnode2$ must become true
Semantics: The event or boolean expression indicated by node $FTnode1$ becomes true in time prior to the time that the event or boolean expression indicated by node $FTnode2$ becomes true. Additionally, $FTnode2$ must become true during the period specified by $t$.

Example: A TCP SYN (connection request) packet must be followed by a SYN ACK packet (connection accepted) within 75 seconds for the connection to continue. A constraint that models this requirement could be:

$$InInterval(TCP\_SYN,TCP\_SYN\_ACK,75\ seconds)$$

### 7.2.3 Generating CPN Templates from SFT

Template Colored Petri Net intrusion detectors may be generated from Software Fault Trees for Intrusions to ensure correctness and correspondence between a requirements specification based on an SFT and a design using CPNs. The constraints added to an augmented SFT to describe the ordering relationships between nodes requires special handling to develop accurate CPN templates from augmented SFTs.

#### 7.2.3.1 Leaf Nodes

Leaf nodes in the augmented SFT for intrusions correspond to basic events in the system which must be detected. Leaf nodes then correspond to token source places in the CPN. The token source places produce a new token each time the basic event takes place. Tokens generated by token source places must have sufficient descriptive information so that tokens may be matched and unified to satisfy any trust, context, and ordering constraints that exist in the augmented SFT.

#### 7.2.3.2 AND Nodes

AND nodes in the SFT are of special interest in intrusion models. Semantically, when all child nodes of an AND node in a SFT are true, the AND node is true.

**AND Nodes without Ordering Constraints** An AND node unconstrained by an ordering in an SFT corresponds to a transition and outgoing place pair in a CPN. An AND node with $n$ inputs translates to a transition with $n$ incoming arcs. Each incoming arc comes from either a token source place of an SFT leaf node, or the outgoing place of an SFT gate node. Figure 7.2 illustrates the
correspondence between an AND node and its equivalent CPN transition place pair, where:

\[
X = \begin{cases} 
1. & \text{if } x \in D_x \\
0. & \text{otherwise} 
\end{cases}
\]

\[
Y = \begin{cases} 
1. & \text{if } y \in D_y \\
0. & \text{otherwise} 
\end{cases}
\]

\[
Z = \begin{cases} 
1. & \text{if } (x \in D_x) \land (y \in D_y) \\
0. & \text{otherwise} 
\end{cases}
\]

Figure 7.2 Unconstrained AND node with corresponding CPN

\(X\) and \(Y\) are the binary inputs to the AND gate, and \(Z\) is the binary output of the AND gate. \(x\) and \(y\) are the incoming tokens to the CPN transition, where \(D_x\) and \(D_y\) are the domains of \(x\) and \(y\), respectively. \(z\) is the output token from the CPN transition.

Tokens leaving the transition must be unified such that they satisfy any trust and context constraints that exist higher in the augmented SFT.

**AND Nodes with Ordering Constraints** Nodes connected to an AND node in an augmented SFT may have an attached constraint that requires the nodes to become true in some particular order. Two cases exist: first, nodes may be required to become true in order, but intervening events may occur; second, nodes may be required to become true in order with no intervening events.

To support ordering, CPN tokens are required to contain times or sequence numbers. If event \(a\) occurs before event \(b\), the timestamp in the token representing event \(a\) must be less than the timestamp in the token representing event \(b\), and no two events may have identical timestamps. Likewise, if event \(a\) occurs before event \(b\), the sequence number in the token representing event \(a\) must be less than the sequence number in the token representing event \(b\), and no two events may have sequence numbers.

Literal wall-clock times used for comparisons are a problem when the times are obtained from different computers in a distributed system [126]. Each computer has its own notion of the current time.
and computer clocks tend to skew at different rates. We assume that the clocks are kept synchronized by the intrusion detection system (synchronized at least once every twenty-four hours), and the skew $\delta$ between a computer's clock and the actual time is very small. As an implementation detail, the intrusion detection system may itself synchronize the clocks and monitor the measured difference $\delta_m$ between clocks. The IDS may have an established maximum skew $\delta_{MAX}$ and may consider any $\delta_m > \delta_{MAX}$ to be an intrusion. In addition, the implementation may include $\delta$ in its comparisons between timestamps. The comparison $t_1 + \delta < t_2 - \delta$ yields a tight bound on two events, which may result in false negatives. The comparison $t_1 - \delta < t_2 + \delta$ yields a loose bound on two events, which may result in false positives.

In the case of sequence numbers, they are maintained per context. No comparison may be made between sequence numbers across contexts.

The addition of temporal ordering to SFT and the associated representation of time information in event tokens enables temporal reasoning.

**Occurs After** The case "occurs after" covers the situation where augmented SFT nodes must become true in a particular order.

Figure 7.3 shows an example of an AND node constrained such that node $y$ must occur (become true) after node $x$ becomes true, where:

$$X = \begin{cases} 
1 & \text{if } x \in D_x \\
0 & \text{otherwise} 
\end{cases}$$

$$Y = \begin{cases} 
1 & \text{if } y \in D_y \\
0 & \text{otherwise} 
\end{cases}$$

$$Z = \begin{cases} 
1 & \text{if } (x \in D_x) \wedge (y \in D_y) \wedge (time1 < time2) \\
0 & \text{otherwise} 
\end{cases}$$

In Figure 7.3, $time1$ and $time2$ denote the timestamps for events $x$ and $y$, respectively. The related CPN segment shows that a token for event $x$ must have a smaller timestamp than the token for event $y$. The significant differences between Figures 7.2 and 7.3 are the addition of time information to the tokens, and the guard on the transition that enforces the ordering on the token's time.

Timestamps in the "occurs after" case may be either wall-clock time or sequence numbers.

**Immediately After** The case "occurs immediately after" covers the situation where augmented SFT nodes must become true in a particular order. Intervening events may not occur.
Figure 7.3 AND node, constrained by "Y after X", with corresponding CPN

Figure 7.4 shows an example of an AND node constrained such that node y must occur (become true) immediately after node x becomes true, where:

\[ X = \begin{cases} 
1. & \text{if } x \in D_x \\
0. & \text{otherwise}
\end{cases} \]

\[ Y = \begin{cases} 
1. & \text{if } y \in D_y \\
0. & \text{otherwise}
\end{cases} \]

\[ Z = \begin{cases} 
1. & \text{if } (x \in D_x) \land (y \in D_y) \land (seq1 + 1 = seq2) \\
0. & \text{otherwise}
\end{cases} \]

In Figure 7.4, seq1 and seq2 denote the sequences numbers for events x and y, respectively. The related CPN segment shows that a token for event y must have the timestamp immediately following the timestamp for event x, implying that discrete timestamps (sequence numbers) are necessary for the operation of this CPN segment.

Figure 7.4 AND node, constrained by "Y immediately after X", with corresponding CPN
7.2.3.3 OR Nodes

When any of the child nodes of an OR node in a SFT are true, the OR node is true.

An OR node in an SFT corresponds to a transition and outgoing place pair in a CPN. An OR node with \( n \) inputs translates to \( n \) transitions, each having 1 incoming arc. Each incoming arc comes from either a token source place based on an SFT leaf node, or the outgoing place based on an SFT gate node. Figure 7.5 illustrates the correspondence between an OR node and its equivalent CPN transitions and place, where

\[
X = \begin{cases} 
1. & \text{if } x \in D_x \\
0. & \text{otherwise}
\end{cases}
\]

\[
Y = \begin{cases} 
1. & \text{if } y \in D_y \\
0. & \text{otherwise}
\end{cases}
\]

\[
Z = \begin{cases} 
1. & \text{if } (x \in D_x) \lor (y \in D_y) \\
0. & \text{otherwise}
\end{cases}
\]

Tokens leaving the transition must be unified such that they satisfy any trust and context constraints that exist higher in the augmented SFT.

Figure 7.5 OR node with corresponding CPN

7.2.3.4 Example: FTP Bounce Attack

The Software Fault Tree in Figure 7.6 represents the combinations of events for an FTP bounce attack to succeed. Deriving a CPN template from the Download-RSH subtree of the fault tree gives a CPN as shown in Figure 7.7. The variables \( \text{seq1}, \text{seq2}, \text{seq3}, \text{and seq5} \) are sequence numbers, and the variables \( \text{time3}, \text{time4}, \text{and time5} \) are times. Figure 7.7 illustrates the use of both sequence numbers and time in the same CPN based on an augmented SFT with temporal constraints.

The use of both sequence numbers and timestamps are necessary for this augmented SFT because there are two separate contexts for events in this tree. The main context is that of the commands and
responses in the FTP session, which are ordered sequentially by sequence numbers. The second context is that of the TCP connection made by the FTP server to the RSH port, whose relationship to the FTP session can only be established by timestamps.

The designer must complete the CPN by augmenting it with:

- Places that provide tokens when events occur in the monitored system
- Information that fully describes the tokens entering and leaving transitions

Figure 7.8 shows the CPN of Figure 7.7 after places have been added that provide the inputs to the transitions.

Figure 7.9 shows the CPN of Figure 7.8 after it has been fully completed with token descriptions using the Design CPN tool [28]. An additional place has been added, named Incoming, to allow this particular CPN to be used as a component in a hierarchical CPN.

7.3 From CPN Design to Distributed Agent Implementation

The Multi-Agent Intrusion Detection System (MAIDS) uses a distributed agent-based system to detect intrusions. If the CPN model of intrusion detection is expanded to include multiple data source nodes (which are simply duplicated places that provide the same token colors to transitions) and transitions are given mobility, the result is a distributed CPN (DCPN).

Previously, section 6.3 examined the implementation of a CPN as Java code. This section further details the implementation of a CPN as agents in a distributed system.

7.3.1 Node Categories

The IDS CPN design resembles a tree where data is obtained at the leaf nodes, fed up through the internal nodes, and finally reaches the root node when an intrusion is identified. Tokens in the IDS CPN represent information that, as tokens “rise” through the tree, is correlated with other information to identify intrusions. Figure 7.10 represents the general CPN for the IDS.

Source places (places which have no incoming arcs) are considered leaf places. The transitions adjacent to leaf places are considered leaf transitions.

Sink places (places which have no outgoing arcs) are considered root places. The Alert place is currently the single root place in the CPN IDS design.

Internal places and internal transitions are the remaining places and transitions, respectively, in the CPN IDS design.
Figure 7.6 Fault tree for FTP bounce attack, with constraints
Figure 7.7 Template CPN from SFT for FTP bounce attack
Figure 7.8  Template CPN for the FTP bounce attack after augmentation with source places
Figure 7.9  Fully augmented CPN for the FTP bounce attack
7.3.2 Leaf Places and Transitions

Raw audit data of various types and formats is obtained from monitored systems for the IDS. Data cleaning agents have been developed to read and process the raw audit data for use by the IDS. The data cleaning agents correspond to the leaf places and transitions in the CPN design.

Leaf places and transitions are duplicated at each monitored system to manage the constant process of data retrieval and cleaning. Figure 7.11 graphically represents the distributed CPN where the leaves are duplicated on each of three monitored systems.

The leaf places (data cleaners) are agents that remain in a single location to obtain raw data, such as that available from log files. In the MAIDS implementation, the leaf places are dispatched by the console to the monitored host, where they will remain stationary for the duration of their activity. The console may also recall the agent to replace it with an updated agent or cease monitoring. The leaf places perform minimal processing and do not place a substantial resource load on the monitored systems.

Leaf places are an instance of places in the MAIDS DCPN implementation that require customized coding. Nearly all other places are passive containers of tokens.

Leaf transitions (data gatherers) are mobile agents that travel between monitored systems to obtain tokens. Figure 7.11 represents the leaf transitions with dashed boxes to indicate that the leaf transitions are mobile. Currently, single instances of each leaf transition perform the data gathering duties, but in
the future, multiple instances of each leaf transition could cooperate to gather data in a large distributed system.

Informally, the leaf transitions perform the first level of data gathering and filtering in the IDS. Formally, the leaf transitions perform the token matching and unification specified by the CPN IDS design.

### 7.3.3 Internal Places and Transitions

Internal places act as passive containers for tokens. Internal places are not duplicated: a single instance exists and accepts tokens from all (possibly mobile) transitions connected to it. Internal places currently reside at the machine running the console, but internal places could be given mobility if it becomes advantageous.

Internal transitions are similar to leaf transitions in that they apply token matching and unification rules to tokens as they are obtained from incoming places and sent to outgoing places. Like internal places, internal transitions are statically positioned at the machine running the console. Internal transitions could be given mobility if advantages are found to such mobility.
7.3.4 Root Place

The root of the CPN IDS design is the alert place. It acts as a passive container, but when a token is added to the alert place, the IDS console interprets the token and displays it. Transitions are required to set an urgency level parameter in tokens for use by the IDS console. Tokens are sorted on the IDS console display by their urgency and then by their arrival time.

7.4 Summary

This chapter details the procedure by which a distributed, agent-based IDS was implemented from a SFT-based requirements and a CPN design. Constraint nodes, specifying trust, temporal, and contextual relationships, are used to augment SFTs and restrict the combinations of events which define intrusions. Algorithmic approaches are used to create CPN templates from augmented SFTs and agent implementations from CPN designs. The end result is an intrusion detection system which detects the intrusions which were specified by the original requirements.

Constraint nodes were added to enable SFTs to model temporal, contextual, and trust relationships between events. Such information is necessary to distinguish actual intrusions from events that bear similarity to intrusions and improve the false-positive rate of the implemented system.

An algorithm is used to convert SFT intrusion specifications into CPN intrusion detector design templates. This conserves the relational constraints of the SFT and preserves the logic of the SFT.

The SFT constraints, SFT to CPN template conversion algorithm, and implementation of the IDS using the CPN design act together to ensure that correctness is preserved from requirements to implementation. The requirements engineer must refine the initial SFT by adding constraints to specify the temporal, contextual, and trust relationships between events that take place as part of intrusions. The designer must finish the CPN design by adding places to provide tokens to the CPN and refining the tokens so that they unify to satisfy the contextual constraints.

Our use of SFT with trust, temporal, and contextual constraints to model intrusions for a requirements specification is novel in the intrusion detection domain. The use of CPNs to model intrusion detection is not new, but the creation of CPNs for intrusion detection from SFT intrusion models is new. Likewise, intrusion detection systems have previous been built using agents, but this is the first use of a CPN model to define the topology of an agent network.
This chapter also detailed the procedure by which a distributed, agent-based IDS was implemented from the CPN design. The implementation preserves the properties of the CPN design while providing an implementation for use on a distributed system.
8 CONCLUSIONS AND FUTURE WORK

8.1 Summary

This dissertation has developed several key ideas for a distributed intrusion detection system based on mobile agents. A flexible, basic architecture for a mobile agent-based intrusion detection system was designed and implemented to demonstrate the operability of such a system, which could be further extended into a complete, operational intrusion detection system based on the modeling work presented in the previous chapters.

A machine learning approach to anomaly detection using system call data from privileged programs was examined. Individual system calls were grouped into windows and then the "bag of words" technique was used to create bitwise feature vectors that described the entire execution of a privileged process. A rule learning algorithm was used to obtain a hypothesis that categorized feature vectors as normal or anomalous. A genetic feature selection technique was used to further improve the accuracy of the rule learning algorithm. Further work along these lines has successfully applied the technique to the Java interpreter to detect anomalous applets [47].

A temporal division of intrusions was presented that splits intrusions into reconnaissance, vulnerability identification, penetration, control, embedding, data extraction & modification, and attack relaying. Such a categorization of intrusions facilitates development of intrusion and intrusion detection models. The temporal categorization also allows an intrusion detection system to relate discrete intrusive events to a single complete intrusion.

Software Fault Tree Analysis was used to model intrusions as combinations of events leading to intrusions. Such a model documents events that lead to intrusions in a such a way that requirements for an intrusion detection system can be based on the SFTA model. The SFTA model results in a convenient, easily-readable specification of what intrusions may be detected, what event data is available to an intrusion detection system, and what combinations of events correspond to intrusions.

Colored Petri Nets were used to model the design of an intrusion detection system. Colored Petri Net designs may be tested and verified against requirements, translated easily to code, and automatically
analyzed for deadlocks & reachability. CPN designs may also be analyzed to determine points at which countermeasures may be applied to stop intrusions in progress. The modularity offered by Colored Petri Nets allows individual intrusion detectors to be composed to form the intrusion detection system.

8.2 Contributions

Specific contributions of this work include:

- Anomaly detection using machine learning and data mining technologies
- Intrusion analysis using Software Fault Tree Analysis to model intrusions
- Creation of IDS requirements from intrusion SFTs
- Analysis of intrusion SFTs to discover countermeasures
- Development of agent-based detection of coordinated attacks by correlating basic attack detections
- Inclusion of data warehouse for data management, data integration, and data reduction & summarization
- Development of a detailed intrusion detection system model using Colored Petri Nets
- Demonstrating satisfaction of SFTA requirements by CPN design through additional constraint nodes in SFT and algorithmic CPN template construction
- Demonstrating satisfaction of CPN design by distributed agent implementation through algorithmic agent construction
- Distributed intrusion detection system implementation using agents based on CPN design model

8.3 Conclusions

An intelligent multi-agent intrusion detection system is convenient, powerful, and dynamically-upgradable. The machine learning approach to anomaly detection has been and will likely continue to be an area of research in the intrusion detection field. The feature vector approach for system call data sufficiently described complete executions of processes such that the RIPPER rule learning algorithm was able to classify anomalies on which it had not been trained. Feature subset selection was employed successfully to further reduce the number of features in the vector.
Software Fault Trees were employed to analyze intrusions. The structured analysis of intrusions enables statement of requirements for an intrusion detection system. The close correspondence between software fault trees as requirements specifications and the design of an intrusion detection system assists development of the design and checking the design for accuracy. Countermeasures may also be identified through analysis of developed software fault trees for intrusions.

Colored Petri Net models enable modeling the design of an intrusion detection system. Such a model may be compared to the requirements for an intrusion detection system to detect faults in the IDS itself. Colored Petri Net models closely correspond to the implementation of the intrusion detection system as agents, reducing the opportunity for errors in implementation.

Algorithmic development of design templates from requirements and implementation templates from designs were examined. If the algorithmic approach correctly maintains the properties of the input, the output templates will automatically satisfy the constraints expressed in the input. The algorithmic approach ensures that the CPN designs maintain the requirements expressed by the SFT-based requirements. Likewise, the algorithmic approach ensures that the mobile agent system faithfully implements the designs expressed as CPNs.

8.4 Future Work

This project has just scratched the surface of several areas of intrusion detection that merit further research. Ideas for future research into a distributed multi-agent intrusion detection system include further incorporation of anomaly detection into the IDS. continue working on the requirements specification and design analysis of the IDS models, improve the efficiency and security of the IDS implementation, and develop support for a widely-applied distributed IDS.

8.4.1 Anomaly detection

The developed SFTA and CPN models for intrusion detection concentrate on misuse detection. Past experience has shown that there will always be new intrusions which do not conform to prior models of intrusions (e.g., intrusions which have yet to be discovered and publicized). It would be necessary to incorporate additional anomaly detection algorithms into the IDS to help discover such intrusions and assist the enhancement of intrusion models to handle the new intrusions. Candidate algorithms for anomaly detectors include frequent episodes [67], association rules, and instance-based learning [65].

Anomaly detection algorithms may misclassify normal activity as anomalous. It would be very useful to develop CPN models that correlate anomalies with related intrusive activity. If an anomaly can not
be automatically correlated with other intrusive activity, it may be less likely to be intrusive.

8.4.2 Requirements specification and analysis

The fault trees developed for requirements specification and analysis are working prototypes. Further research, development, and analysis of the SFTA model of intrusions will help answer questions including:

- Is the SFTA technique ideal?
- Are there alternative specification techniques that work better?

8.4.3 Design using CPNs

A few examples of misuse were selected for modeling using CPNs. It would be useful to develop a more complete CPN design for the intrusion detection system based on a full set of requirements specifications to determine how well the requirements specification documents the system and how well the CPN design models the IDS.

The correspondence of the requirements specification to the CPN design will help demonstrate the correctness of the CPNs given the requirements specification.

The CPN design needs to be enhanced to include anomaly detection. Methods of developing models of anomaly detection need to be investigated. Anomaly detection models need to be developed. Comprehensive models that correlate anomalies with other anomalies and misuses also need to be developed and analyzed.

8.4.4 IDS Security and Performance

The prototype intrusion detection system that was developed as part of this project is implemented in Java using the Voyager Object Request Broker. Any intruder with sufficient privilege may kill the agent server or interfere with agent communication. The IDS requires a fair bit of CPU power and memory, which may be an unacceptable load on production systems.

It would be necessary to improve the security of the IDS such that it would be resistant to tampering. Ideas for hardening include:

- Moving parts of the IDS into the operating system kernel;
- Moving the IDS into a process specially protected by the kernel;
• Securing communication between agents to prevent spoofing, modification, and eavesdropping.

Performance improvements include moving parts of the IDS from an interpreted language to a compiled language. It may be sufficient to compile the existing Java code into machine code instead of the interpreted byte code.
BIBLIOGRAPHY


